UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Energy-efficient architectures for recurrent neural networks

Franyell Silfa

Energy-Efficient Architectures for Recurrent Neural Networks

Franyell Silfa

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Doctor of Philosophy

Department of Computer Architecture
Universitat Politècnica de Catalunya

Advisors: José-Maria Arnau, Antonio González

October, 2020
Barcelona, Spain

Abstract

In recent years, Deep Learning algorithms have been remarkably successful in applications such as Automatic Speech Recognition, Sentiment Analysis and Machine Translation. As a result, this kind of applications are ubiquitous in our lives and are found in a plethora of devices extending from small IoT systems to cloud servers. However, even for edge devices, the evaluation of Deep Learning algorithms is typically offloaded to the cloud. The main reason is that these algorithms are composed of Deep Neural Networks (DNNs), such as Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs), which have a large number of parameters and require a large amount of computations. Therefore, deploying DNNs on devices with a constrained energy and memory budget is a challenge since the memory and power requirements of DNNs are extremely large.

RNNs are usually employed to solve sequence to sequence problems such as Speech Recognition and Machine Translation. In addition to having extremely large memory and power requirements, they also contain data dependencies among the executions of time-steps, and thus the amount of parallelism is severely limited. Therefore, the challenge to evaluate them in an energy-efficient manner is higher than the evaluation of other DNN algorithms, such as CNNs. This thesis aims to study applications using RNNs to improve their energy efficiency on specialized architectures. More specifically, we propose novel energy-saving techniques and highly efficient customized architectures tailored to the evaluation of deep RNN models. There are several RNN topologies, but the most successful are the Long Short Term memory (LSTM) and the Gated Recurrent Unit(GRU). Consequently, they are the main focus of this work.

To carry out our objective, as an initial step, we characterize a set of state-of-the-art RNN workloads running on a modern SoC. We identify that accessing the memory to fetch the network parameters is the major source of energy consumption accounting for up to 80%. Moreover, we observe that real-time evaluation requires a high power budget. Hence, we propose a novel energy-efficient processing unit to perform RNN inference, which we call E-PUR. By employing a cycle-level simulator, we show that E-PUR achieves 6.8x speedup and a reduction on energy consumption of 58x when compared to a modern SoC. We further improve E-PUR's energy efficiency by developing a novel technique that maximizes the temporal locality of fetching the network parameters, which we call Maximizing Weight Locality (MWL). The final proposal reduces energy consumption by 88x on average, while the power dissipation is 10x lower.

Although E-PUR's energy efficiency is extremely high, we observe that fetching the parameters continues to be the primary source of energy consumption. Hence, we strive to reduce memory accesses and propose a novel technique that reuses previously cached computations. Our key observation is that when evaluating the large input sequences (audio frames, words, etc.) of an RNN model, the output of a given neuron tends to change lightly between consecutive evaluations. Therefore, we exploit this observation to develop a hardware scheme that caches the neurons' outputs and reuses them whenever it detects that the change between the current and previously computed output value for a given neuron is small. Consequently, we do not fetch the weights or perform the computations for that neuron. One of the main challenges is implementing a low-cost hardware solution to predict when a neuron's output for the current evaluation and the previously

cached result will be similar. We address this issue by employing a Binary Neural Network (BNN) as a predictor of reusability. The low-cost BNN can be employed in this context since we show that its output is highly correlated to the output of RNNs. In other words, when the output of the BNN changes lightly between consecutive evaluations, it is very likely that the same happens for the output of the RNN. We show that our proposal avoids more than 24.2% of computations. As a result, on average, energy consumption is reduced by 18.5% while achieving a speedup of 1.35x.

The memory footprint for RNN models ranges from a few megabytes to hundreds of megabytes, and usually, their size is reduced by using low precision for its evaluation and storage. In this regard, the minimum precision used is identified offline by an static profiling of the model. Also, it is set such that the model maintains its original accuracy. We observe that this method utilizes the same precision to compute all time-steps. However, our experiments show that some time-steps can be evaluated using a lower precision while preserving the original accuracy. Therefore, we propose a novel technique that dynamically selects the precision employed to compute each time-step. A significant challenge of our proposal is deciding when to choose a lower bit-width. We address this issue by recognizing that information from a previous evaluation can be employed to determine the precision required in the current time-step. Specifically, we use the cell state and the cell output for LSTM and GRU models, respectively. By selecting the precision based on previous time-step information, our scheme evaluates 57% of the computations on a bit-width that is lower than the fixed precision employed by methods using the same bit-width during all the time-steps. We implement our proposal on E-PUR and evaluate it for a set of modern RNN networks. We show that it provides 1.46x speedup and 19.2% energy savings on average without any accuracy loss.

Our previous proposals are tailored to systems that execute one input sequence at a time. However, systems such as cloud services batch several input sequences together and compute them simultaneously. Hence, as a final optimization, we focus on batching for RNNs. RNN batching is employed to increase throughput. However, it requires using a large amount of padding due to the variability in length (i.e., number of time-steps) of the batched sequences. Hence, some schemes use a fine granularity (i.e., a small number of time-steps) when batching aiming to reduce the amount of padding. However, they perform a large number of parameters swap. The reason is that each time an RNN layer is processed, a new set of weights is loaded into on-chip memory. Henceforth, weight reuse is severely affected in previous RNN batching proposals. As a result, their energy efficiency is extremely low. We propose E-BATCH to address these issues. E-BATCH is a low-latency and energy-efficient batching scheme tailored to RNN accelerators. It requires a runtime system and simple hardware support in the RNN accelerator. The runtime is in charge of concatenating the input sequences so that batches with a large number of time-steps are created. Therefore, weight reuse is increased. On the other hand, we extend the hardware accelerator so that it tracks the state of the requests while they are evaluated. Also, it notifies the runtime once a request is completed, hence it allows a new input sequence to join the ongoing evaluation of a batch, effectively decreasing the amount of padding required.

Finally, we evaluate E-BATCH on top of two state-of-the-art RNN accelerators, E-PUR and the Tensor Processing Unit (TPU). We show that, compared to the state-of-the-art RNN batching techniques, E-BATCH improves throughput by 1.8x and energy-efficiency by 3.6x in E-PUR, and by 2.1x and 1.6x, respectively, in the TPU.

Keywords

Deep Learning, Deep Neural Networks, Recurrent Neural Networks, Convolutional Neural Networks, Hardware Accelerator.

Acknowledgements

First of all, I want to express my sincerest gratitude to my advisors, Jose-Maria Arnau and Antonio González, for their continuous support during my Ph.D. study and related research, for their patience, motivation, and tremendous knowledge. I sincerely believe that I could not have better advisors. I am very grateful to Prof. Antonio González for giving me the opportunity to join the ARCO group and learn from him. I appreciate your invaluable guidance, your timely feedback, and your always wise advice. Thanks for pushing me to be the best and teaching me the importance of doing high-quality research.

I will always be in debt with Dr. Jose-Maria Arnau for all your support and faith in me. I appreciate everything that you taught me and all the discussions that we have throughout these years. Thanks for your invaluable guidance and fruitful ideas. I will never forget your encouragement and constructive comments. Your patience and persistence made completing this Ph.D. possible.

I owe a tremendous debt to La Fundación Carolina and PUCMM for providing me financial support during these years. I am also very grateful to Roberto Abreu, Victor González, Julio Ferreira, and Jesús Ramírez, which have always been pending about my welfare and success during the Ph.D.

I want to make a special mention to Gem Dot, who helped me a lot at the beginning of the Ph.D. Thanks for your wise advice, support, and inspiring discussions about computer architecture and life.

I want to thank all the members of ARCO that I met over these years. I was fortunate to be part of a well-known and enriching research group. Special thanks to Enrique, Hammid, Reza, Marti, Marc, Albert, Azue, Josue, Dennis, Diya, Raul, Pedro, Jorge and Mehdi for the stimulating discussions, the coffee breaks, and for all the great moments that we had at UPC. Also, thanks for your support and encouragement after those endless paper rejections.

I feel blessed to have amazing friends in my life. Especially, I would like to express my appreciation to Reyna, Luis (kp), Billy, Leandro, Andres, Manuel, Junior, Engerst, and Aby. Thanks for your friendship, motivation, and moral support. It made this journey more comfortable, and I am very thankful for that. Also, I would like to thank my second family in Barcelona. Mara, Carolina, Fifi, and Aby, thank you for worrying about my well-being and opening your doors and hearts.

Finally, I owe my deepest gratitude to my family for their unconditional love and support. Thanks to my brothers and sisters, Fabiannys, Josefina, Fabiolys, Heydi, and José for their continuous encouragement, and spiritual support. Particularly, I would like to thank my brother Fabio Antonio and his wife, Maria Luisa, for their unconditional support and caring. Also, I am grateful to my cousins, aunts, and uncles. I will always be in debt with my parents Fabio and Mercedes, for their genuine love, patience, and support. Especially, I will like to thank my mother for being a role model, her hard work, and always caring and supporting my brothers and me. I have the most profound appreciation for her prayers and unconditional love.

This thesis is dedicated to my grandmother Josefa
who always encouraged me to study, be myself,
and taught me the importance of being a good person.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents the motivation behind this work and explains the challenges of performing RNN inference in hardware accelerators in an energy-efficient manner. It then provides a summary of the related work and an overview of the main contributions of this thesis. Also, we highlight the key differences between our solutions and the state-of-the-art approaches.

1.1 Motivation

Deep learning algorithms have become ubiquitous in many computer science domains. They are employed on a plethora of commercial and academic applications such as image classification [60, 50], speech recognition [7, 69], machine translation [110, 58], video classification [113, 91] and sentiment analysis [23, 13]. Moreover, they are applied in less mainstream applications such as the smart toilet [76] or to play video games [103]. Not surprisingly, evaluating deep learning algorithms on cloud servers and low power devices, in an energy-efficient manner, has become a relevant and very active research area.

Evaluating deep learning applications in cloud servers is very important since many companies provide online services powered by machine learning algorithms. Services such as conversational agents (chatbots) [4, 27] and grammatical analysis [39] are generally evaluated in the cloud. Moreover, applications such as semantic search [67], which is employed by banks to navigate through customer documents, and sentiment analysis [96] for brand management, are better suited to the cloud computing domain since their main target is the enterprise sector.

Deep learning applications are also very relevant on low power mobile devices such as Smartphones, Smartwatches and IoT devices. On Smartphones, they can be used to create more natural human-machine interfaces that are controlled by voice [10, 36]. Additionally, real-time speech trans-

lation allows conversations between people speaking different languages. On Smartwatches, they are used to identify activities such as eating or drinking habits [108]. Finally, deep learning enables IoT devices to react and intelligently process information [100].

Deep Neural Networks (DNNs) are the main reason for the tremendous results obtained by deep learning algorithms. DNNs are composed of millions of neurons and the connections among them (i.e. synaptic weights) [65]. These weights determine the importance of the information that is being processed. Moreover, they are computed and adapted to the application domain using a training algorithm.

The number of applications of deep learning algorithms on cloud servers and low power mobile devices are countless. Hence, evaluating them in an energy-efficient manner is essential to decrease inference services' prices on the cloud. Moreover, high energy-efficiency is required to perform inference in low power mobile devices. In this regard, the evaluation of deep learning applications requires a humongous amount of memory and power, which hinder their implementation on mobile devices. For this reason, DNNs are usually evaluated in the cloud using specialized hardware architectures (i.e., accelerators) such as (TPU [51], Brainwave [29]), GPUs, or FPGAs. Nevertheless, this approach has some drawbacks. First, it requires a permanent internet connection. Therefore, it is impossible to use them if an internet connection is not available. Second, the transmission to the cloud increases the latency, which may result in unacceptable delays that violate real-time constraints. Moreover, due to the high power dissipation, performing inference on the cloud server is very costly. Finally, offloading the processing to a cloud server might result in larger energy consumption. The reason is that the communication energy per byte can be thousands to millions of times bigger than the energy per instruction [85].

There are many topologies of DNNs. However, the most common are Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs are employed on problems such as image classification [60] and human pose recognition [101]. On the other hand, RNNs have achieved tremendous success for a wide variety of sequence-to-sequence application domains [110, 113, 23, 7, 69, 54]. Recent data presented in [51] shows that RNNs account for 30% of machine learning workloads in Google's data centers, whereas CNNs only represent 5% of the applications. The major reason for the success of RNNs is that they have loops that allow them to use information from previously processed inputs. Also, they can handle problems with variable input and output length.In this work, we focus on RNNs. More specifically, the main objective is to improve the energy-efficiency of RNN inference in hardware accelerators, while also improving their performance.

The memory footprint and power requirements of RNNs are extremely large. Figure 1.1 shows the memory footprint for several state-of-the-art applications employing RNNs. As it can be seen, their sizes range from a few hundreds to several megabytes, making its deployment in small IoT devices a challenge since their memory footprint is usually in the order of kilobytes or a few megabytes. Furthermore, due to its recurrent nature, RNN inference exhibits a significant amount of sequential processing and limited parallelism. As a result, it cannot be efficiently executed on multicore CPUs, GPUs or state-of-the-art high performance accelerators for DNNs.

We evaluate several RNNs applications in a modern SoC (i.e., Tegra X1) [72] and an accelerator for RNNs. Figure 1.2 shows the power consumption for several RNN applications running on these devices. For the Tegra X1 the power consumption is 4.3 watts on average. This power dissipation is

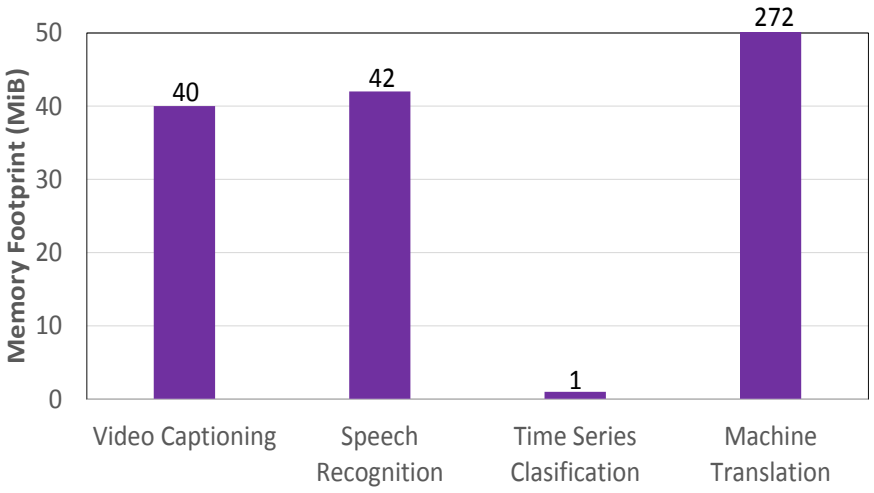


Figure 1.1: Memory Footprint for several RNN applications.

not amenable for low power devices that are operated from a battery source. On the contrary, the power consumption of the accelerator is 330 milli-watts on average, and thus the power is reduced by 10x on average.

Some applications, such as speech recognition, in addition to high accuracy also require real-time evaluation to maintain user satisfaction. However, real-time inference comes at a high energy cost. For instance, for the speech recognition network shown in Figure 1.2, the power dissipation of the SoC is extremely high, our measurements show that it does not achieve real-time performance. More specifically, the SoC takes 2.72 seconds to decode one second of speech while its power dissipation is 5.6 watts. On the contrary, it takes 0.03 seconds to decode the same audio in the accelerator and the power dissipation is 0.3 watts.

Energy-efficient evaluation of RNNs in cloud servers is also challenging. In this regard, state-of-the-art accelerators such as TPU [51] and Brainwave [29], and high-end GPUs are employed to evaluate large RNN models while providing high-performance. These accelerators are generally composed of a systolic array consisting of a large number of processing elements (i.e., mac units). Due to the recurrent nature of RNNs, many computations must be serialized during their evaluation. As a result, the amount of parallelism available is limited, and processing elements are underutilized. For instance, the resource utilization for RNNs is: 18% and 3.5% for TPU and Brainwave, respectively. Also, resource utilization is moderate for GPUs. For instance, a high-end GPU (Titan V) has resource utilization ranging from 4% to 28% [112]. Besides the low resource utilization, these accelerators have a power dissipation in the order of hundreds of Watts, which increases the price of evaluating inference services on them. For instance, TPU V2 [37] and Brainwave's power dissipation is 200W and 125W, respectively, whereas it is in order of hundreds of watts for high-end GPUs.

As mentioned earlier, performing RNN inference on either cloud servers or edge devices is very challenging. For edge devices, this difficulty is due to their hardware limitations. On the contrary, offloading RNN inference to cloud servers has a high cost in performance and energy. Besides, for the high-performance accelerators commonly found in the cloud, resource utilization is very low while power dissipation is high. As shown earlier, a hardware accelerator for RNN can reduce the

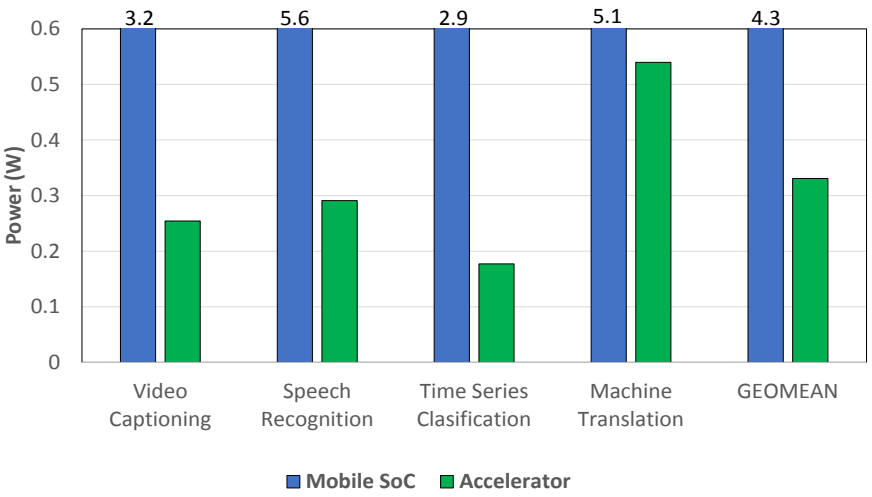


Figure 1.2: Power dissipation for several RNN applications on a modern SoC and an accelerator.

power consumption during RNN inference to milli-watts, which is amenable for cloud severs and low power devices. Therefore, we claim that hardware acceleration and energy-saving techniques are crucial in implementing RNN inference in an energy-efficient manner. Consequently, we propose a novel, highly-efficient custom architecture tailored to RNN inference. Moreover, we design several novel energy-saving techniques that further enhance the energy efficiency of our proposal.

1.2 Problem Statement, Objectives and Contributions

RNNs are usually employed to solve sequence to sequence problems. In this context, an input sequence is passed to an RNN, and then it outputs the most likely sequence according to a previously learned probability distribution. Typically, an RNN is composed of millions of weights which are used during inference to compute its final output. Therefore, as outlined before, its memory footprint and power requirements are significant. Furthermore, its output is fed back to the input allowing past information to persist from one execution to the next. Hence, the evaluation of an input sequence in an RNN is strictly sequential due to the data dependencies. For this reason, an RNN cannot be efficiently executed on state-of-the-art accelerators, modern multicore CPUs or GPUs. We present detailed background information for RNNs in Chapter 2.

Due to the importance of evaluating large and small RNN models in hardware accelerators, either in edge devices or in the cloud, the challenges that it carries, and the significant benefits of using an architecture tailored to RNN inference, we concentrate this thesis on hardware accelerators for RNN inference.

The main objective of this thesis is to perform RNN inference on hardware accelerators in an energy-efficient manner while also improving performance. We address the problems that hinder the evaluation of RNNs efficiently by proposing novel techniques and custom architectures tailored to improve their performance and energy efficiency. First, we tackled the issue of efficiently evaluating large RNN models by offloading their execution to a custom architecture, which does not need the entire RNN model on local memory during its evaluation. Second, we overcome the problems of low

performance and high power dissipation during RNN inference by employing a highly optimized pipeline tailored to RNN computations and by improving the temporal and spatial locality of the RNN's parameters. Third, we further enhance the performance and energy efficiency of the custom architecture by proposing two novel techniques that employ memoization and reduced precision to avoid computations and increase weight reuse. Finally, we approach the problem of low energy efficiency during the evaluation of several input sequences concurrently (i.e., RNN batching) by proposing a novel batching system that trades latency for increased weight reuse.

The following sections outline the problems we are trying to solve, describe the approach we take to solve them, and highlight the novel contributions of this thesis.

1.2.1 RNN Inference on an Energy-Efficient Processing Unit

Figure 1.1 shows the memory requirements of some important RNN applications. As it can be seen, the memory requirements for some models are significant, and the models cannot be deployed entirely on small IoT devices, which generally only have a few megabytes of memory storage. Note that state-of-the-art architectures for RNN inference [41, 62, 61] usually include local on-chip memories to improve energy efficiency, and they keep the entire RNN model locally. While this approach is very efficient energy-wise, it is only applicable to small RNN models. Consequently, an important class of RNN applications such as machine translation and speech recognition cannot be evaluated on these architectures. Other state-of-the-art proposals [17, 31, 43] mitigate the issue of high storage requirement by compressing the model employing reduced precision and/or pruning [43]. While these proposals support large RNN models, their approach degrades the model's accuracy (i.e., the ability to predict correctly decreases).

To address the problem mentioned above, we observe that only one RNN layer[1] needs to be kept on local on-chip memory at any given time since values (i.e., weights) are not reused among RNN layers. Consequently, we alleviate this problem by proposing to keep only one RNN layer on local on-chip memory during inference. This approach effectively reduces the required on-chip memory to the storage requirement of one RNN layer, which is typically not more than a few megabytes. Furthermore, the current trend is to create RNNs with several layers to increase their accuracy [26]. Therefore, the proposed solution also scales-up with the current trend.

Figure 1.2 shows the power dissipation for several RNN applications running on a modern SoC. As can be seen, the power dissipation is 4.3 watts on average, which is not amenable to low power devices. Moreover, state-of-the-art proposals for RNN inference that can evaluate large RNN models have a power dissipation ranging from 10 to hundreds of watts [41, 62, 43, 37, 29]. To illustrate this problem from a user's perspective, consider a Smartphone [9] whose battery has a capacity of 11.91 Wh. Also, assuming that it employs the RNN application for speech recognition, shown in Figure 1.2, to create a better human-to-machine interface. In this case, the power dissipation is 5.6 watts, and as a result, the operating time will decrease to 2.7 hours. Besides, the SoC has a poor performance when evaluating this model: it takes 2.72 seconds to decode one second of speech. Hence, it does not achieve real-time performance which is a requirement to ensure user satisfaction. Note that since the SoC is tailored to a broader range of applications, its performance is impacted

[1] In this context, an RNN layer can be viewed as an array of weights.

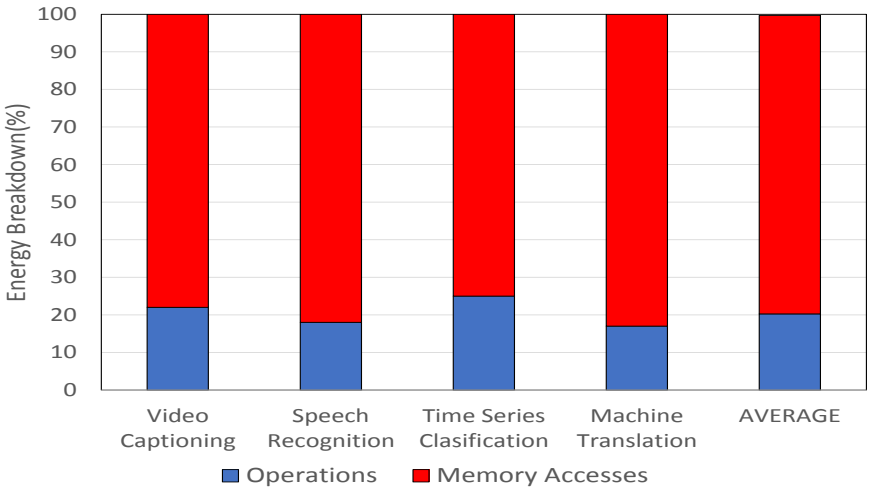


Figure 1.3: Energy breakdown for several RNN applications in a state-of-the-art accelerator.

by the overhead due to related tasks (e.g., GPU synchronization, CPU work, etc.). Moreover, due to the RNN data dependencies, the hardware resources on the SoC are not fully utilized.

To overcome the problem of high power dissipation in state-of-the-art solutions for RNN inference, we observe that most of the energy consumption is due to memory accesses to fetch the model parameters (i.e., weights). As shown in Figure 1.3, the energy consumption due to memory accesses is close to 80% on average. We solve this problem by offloading RNN inference to a hardware accelerator, that we call E-PUR, and applying several architectural optimizations to it.

RNN inference involves performing several matrix-vector multiplications. Consequently, E-PUR is composed of an extremely optimized pipeline to compute matrix-vector multiplications and activation functions (i.e., hyperbolic tangent). Also, it includes several on-chip memories to store one RNN layer during inference. Hence, E-PUR solves the issue of evaluating large RNN models. Moreover, it includes several on-chip memories so that when values are reused, the high energy cost of accessing the main memory is mitigated by accessing the local on-chip memories. As a result, energy consumption is reduced.

In E-PUR, we approach the problem of poor performance presented in the mobile SoC by hiding memory latency (i.e., loading/storing is overlapped with computations) and reducing memory traffic. Moreover, due to the custom pipeline tailored to RNN computations, several computations that do not have data dependencies are evaluated in parallel.

Our evaluations show that E-PUR reduces execution time by 6.9x, on average, compared to a mobile SoC [72]. Moreover, energy consumption is reduced by 58x on average, while the power dissipation is 500 mW. Although these results significantly improve the performance and energy consumption of RNN inference, we observe that the design can be further improved. More specifically, we observe that the RNN computations can be divided into calculations due to the input sequence and calculations due to the previous output. The computations due to the input sequence can performed in parallel, while computations that include the previous output must be evaluated sequentially. Hence, we first perform the computations involving the input sequence in parallel, and then we evaluate the computations for the previous output sequentially. We call this

technique Maximizing Weight Locality (MWL). By employing MWL, the size of the local on-chip memories to store the weights is reduced by 2x, since only the weights corresponding to the input sequence have to be kept locally on-chip memory. E-PUR and MWL are described extensively in Chapter 4. This work has been published in the proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT) [93].

In summary, we present E-PUR: an energy-efficient processing unit for RNN inference, whose foremost goal is to solve the problem of evaluating large RNN models in an energy-efficient manner while also improving performance. We overcome this problem by only storing one RNN layer locally on-chip memory, by employing a highly optimized pipeline, and by maximizing the temporal locality of the weights. It improves performance and energy consumption by 6.9x and 88x on average, respectively. Moreover, it has an average power dissipation of 330 mW.

1.2.2 Energy Saving Techniques

Although E-PUR is highly optimized and provides substantial energy savings, most of its energy consumption is still due to accesses to main memory and/or to the local on-chip memories. We further address the problem of energy consumption during RNN inference by avoiding memory access and computations. To achieve this, we propose the following two novel techniques. The first one reuses previous results to avoid memory accesses and computations, whereas the second one avoids fetching large amounts of data by dynamically reducing the bit-with used to perform the computations.

Neuron-level Fuzzy Memoization

In RNNs, an input sequence is evaluated sequentially for each of its elements (i.e., time-step). Similarly, an RNN gives an output for each time-step of the input sequence. Also, the output of a given RNN layer is composed of the output of all the neurons in it. In this context, a neuron is a functional unit whose output is a weighted sum between an input time-step and the neurons' weights.

By monitoring the output of multiple neurons during the evaluation of several time-steps, we observe that the output of a given neuron tends to change lightly between consecutive evaluations. Specifically, we note that the relative difference between the previous and current output of a given neuron is smaller than 23% on average. Additionally, previous work had reported [115] that RNNs are inherently error-tolerant. Therefore, based on these two observations, we aim to further improve energy efficiency by avoiding memory accesses and computations. To accomplish this, we propose to cache and reuse the previous output of a neuron.

As an initial step, we evaluate the impact on accuracy and the potential of reusing previously cached computations. For this, we use an oracle predictor that always predicts the correct output value for a given neuron (i.e., it computes the neuron's output on each time-step). Then, we implement a functional scheme where a previously cached value is used when the relative difference between it and the value predicted by the oracle is less than a fixed threshold. We evaluated this

scheme on several RNNs, and it revealed that the potential for reuse is between 50% and 20%, while maintaining the base accuracy of the RNN.

Secondly, since the oracle predictor has to perform the actual computations, we approach the problem of implementing and replacing it with a low-cost energy solution. To solve this issue, we employ a Binary Neural Network (BNN) as a reusability predictor, and the rationality behind this decision is the following. We found that if we derive a BNN as a mirror of a full-precision RNN, then the output of a given neuron in the BNN and the output of the corresponding neuron in full-precision are highly correlated. Therefore, if the output of the binary neuron changes lightly, it is highly likely that the full-precision neuron's output will exhibit the same behavior. We show in Chapter 5 that the BNN provides excellent results when used as a predictor of reusability. Finally, one advantage of using BNNs is that they are hardware friendly since weights are stored using only one bit, and their computations are performed using NOR gates [80].

In the third place, we develop a fuzzy memoization scheme and implement it on top of E-PUR. Our experimental results show that our technique achieves 24.2% of computation reuse on average. Consequently, the energy consumption is reduced by 18.5% on average, while obtaining a 1.35x speedup. This technique is detailed in Chapter 5. Finally, this work has been published in the proceedings of the 52th International Symposium on Microarchitecture (MICRO) [94].

Dynamic Precision Selection

Traditionally, RNN models are trained and stored using full-precision (e.g., fp32) values for the weights and input sequences. Moreover, as mentioned earlier, the memory requirements of RNNs tend to be very large, so reduced precision (i.e., 8-bits) is usually employed to compress RNN models. Typically, when reduced precision is applied, the parameters and inputs are stored as integer values, while the computations are done using integer arithmetic [51, 19, 43]. This approach significantly lowers the amount of energy required for computations and memory accesses.

Regarding the precision (i.e., bit-width) employed to compress an RNN model, state-of-the-art solutions, such as ESE [43], TPU [51], and Eyeris [19], use a fixed precision to store and evaluate an entire RNN model, which is determined offline by extensively profiling the RNN model. Commonly, when a fixed precision is used, a worst-case bit-width is chosen, such that the model accuracy is maintained. However, the precision requirements for different RNN layers are not always identical [52]. State-of-the-art proposals such as Stripes [52] and Bit-Fusion [92] employ a variable precision to improve performance and energy efficiency further. In these proposals, the precision used to evaluate a given RNN layer is determined by its requirements and its impact on the model accuracy.

Despite the additional flexibility of accelerators supporting variable precision among layers, the precision for each RNN layer is determined offline and is fixed. However, in our experiments, we have observed that for RNNs, the precision requirements of a given RNN layer are also different when evaluating some time-steps. More specifically, we experimentally confirmed that for some time-steps, we could use an even lower precision based on previous information (i.e., the cell state which is defined formally in Chapter 2) produced by the RNN layer. Therefore, we exploit this observation by proposing a novel technique that, at runtime during RNN inference, selects a lower

precision for some computations. In contrast, the rest are evaluated using a higher bit-width.

Regarding the behavior of the cell state, we profiled and analyzed it after evaluating multiple time-steps on several RNN models. After this profiling, we observe that when the value of the cell state is inside certain regions (i.e., *stable regions*), a lower precision could be employed without degrading the original accuracy of the RNN model. On the other hand, when the value of the cell state is outside the *stable region*, we employ a higher precision. Note that, when a lower precision is employed, the amount of data that needs to be fetched is smaller, and thus energy consumption is improved, which is our primary objective. Moreover, computations can be performed faster, assuming that the required supporting hardware is provided.

Based on the previous observation, we design and propose a hardware mechanism that tracks the cell state, and based on its value: a lower or higher precision is selected. Our scheme is mainly composed of buffers to store meta-information about the cell state. Our experimental results show that our proposal selects a low precision for 57% of the computations without degrading the accuracy. Therefore, our scheme is very beneficial, yet it only requires small changes to the underlying accelerator.

Finally, we implement our proposal on top of E-PUR and employ hardware multipliers (serial and parallel) that support variable precision. Our evaluations show that the performance is improved by 1.48x on average. Moreover, energy consumption is decreased by 19.2% on average, while the area overhead is 8%. Currently, this work is under revision for publication, and it is described extensively in Chapter 6.

Energy-Efficient RNN Batching

The amount of parallelism during RNN inference is limited due to the recurrent nature of RNNs and their strict data dependencies. For this reason, resource utilization is low on many state-of-the-art systems. For instance, TPU and BrainWave have a resource utilization of 18% and 3.5%, respectively. E-PUR has almost 100% resource utilization, but it can only evaluate one input sequence at the time. Some systems for RNN inference, such as cloud servers, increase energy efficiency, parallelism, and throughput by batching several RNN input sequences so that they are evaluated concurrently. However, batching RNN sequences is difficult since usually batching requires that all the batched sequences have an identical length (i.e., number of time-steps), which is not the case for RNN. The standard solution to solve this problem is to employ padding [2]. However, we observe that padding incurs in a high amount of energy consumption. Moreover, state-of-the-art batching techniques for RNNs, such as cellular batching [33], are very inefficient energy-wise.

As a first step, we analyze and identify the primary sources of energy inefficiencies during RNN batching to overcome the problem of high energy consumption. Our evaluations show that when padding is employed during RNN batching, 30.2% of the energy consumption is caused by the computations added by padding. Furthermore, the latency overhead is 28.5% on average. Moreover, we observe that state-of-the-art batching proposals such as Cellular Batching [33] tend to fetch an RNN layer several times to evaluate a given input sequence. As a result, the temporal and spatial locality of the weights is severely affected.

After identifying the primary sources of energy inefficiency during RNN batching, we develop a novel batching scheme which reduces the amount of padded computations while maintaining the temporal and spatial locality of the weights. Our system distributes the input sequences among the available hardware units (e.g., matrix-vector multipliers) such that the amount of padding is minimized. Also, when hardware units became available, the input sequences that are available for execution are sent to those units, and thus increasing weight reuse. Finally, since creating batches with short sequences on them provokes a large number of weights swaps, our scheme strives to create batches with long sequences, which is achieved by merging several input sequences into one long sequence. Our proposal includes a runtime system which creates and tracks the state of the batches and the sequences on them. Also, we add small yet effective modifications to the underlying hardware accelerator.

Finally, we implement our proposal on top of an E-PUR-like and TPU-like architecture. Our evaluations show that energy efficiency is improved by 3.6x/2.1x on average for E-PUR and TPU, respectively, whereas throughput is enhanced by 1.8x/1.6x on average. Most of the hardware structures needed by our proposal are small buffers, which have a low energy cost. Currently, this work is under revision for publication, and it is detailed in Chapter 7.

1.3 State-of-the-art in Energy-Efficient RNN Inference

In recent years, improving the energy-efficiency of RNN applications during inference has attracted the attention of the computer architecture community. As we mentioned earlier, hardware acceleration is a critical factor in achieving high energy-efficiency. Not surprisingly, a plethora of custom architectures have been proposed to improve the energy-efficiency of RNN inference. Furthermore, techniques such as pruning, compression, and quantization are employed to decrease the RNN memory requirements and increase energy-efficiency. In this section, we present several hardware accelerators for RNN, and we detail some of the existing energy-saving techniques.

1.3.1 Hardware Accelerators for RNN

Several works aiming to optimize RNN inference have been proposed for different hardware platforms: GPUs [24, 11], FPGAs [41, 62, 43, 29, 107, 31, 61], and ASICs [51, 112, 42]. One of the most significant drawbacks of these proposals is that their power dissipation is not amenable to edge devices. Moreover, some proposals are limited to small RNN models.

GPUs

GPUs are one of the most common platforms employed to perform RNN inference. Thus, some recent proposals strive to improve their performance [24, 11] for RNN inference. In this regard, RNN models are implemented using software frameworks such as Tensorflow [2] and PyTorch [79] which usually employ software libraries such as cuDNN [11] or cuBlas [1] as their backend. These libraries are highly optimized to perform matrix-vector multiplication on GPUs efficiently. However,

their performance is severely affected during RNN inference due to the data dependencies. Also, these libraries are optimized to compute batches with a large number of RNN sequences during training. However, the batch size during RNN inference is usually small (i.e., one). Therefore, they are unable to use all the resources available on the GPU. Furthermore, during RNN inference the power dissipation of high-end GPUs is in the order of hundreds of watts [43, 25, 117], whereas during RNN inference it ranges from 5 to 12 watts [32, 17] for mobiles GPUs.

Although RNN inference on GPUs outperform CPUs, they are not amenable to low-power devices due to their high power dissipation. In this thesis, we address this issue by offloading RNN inference to an accelerator since the architecture can be fully customized for this problem. Not surprisingly, our experiment shows that for several RNN applications, energy efficiency is improved by 88x when inference is offloaded to E-PUR (our proposed accelerator) compared to a mobile GPU which employs cuDNN [11] as a backend. Also, the performance is improved by 6.8x, while the accelerator's power dissipation is 330 mW on average. Moreover, when cuBLAS [1] is employed as a backend, the energy consumption is reduced by 92x, and a speedup of 18.7x is obtained.

FPGAs

FPGA-based accelerators for RNN inference have been proposed in [41, 62, 43]. While these proposals achieve better performance per watts than GPUs, their power dissipation ranges from 19W to 41W. On the contrary, E-PUR has an average power dissipation of 330 milli-watts.

Solutions targeting low power dissipation have also been proposed. The works presented in [17] have a power dissipation of 1.9 W, whereas it is 5.5 W for the accelerator in [31]. A significant drawback of these proposals is that they are tailored to small RNN models (i.e., one RNN layer and less than 256 neurons [2]). Besides, both of these works use fixed-point arithmetic for computations and Q8.8 [3] as the data format. Consequently, the model accuracy decreases by 7.1% in [17] and by 2% in [31]. Scaling up those accelerators to support larger models incurs in a high energy cost due to the increased traffic to local memory storage and/or accesses to the main memory. In this thesis, we solved this problem by only storing one RNN layer locally in on-chip memory. For instance, E-PUR supports large RNN models (i.e., 2048 neurons), and it can also scale up to support model with a large number of RNN layers.

The RNN accelerator presented by Lee et al. [61] is tailored to speech recognition and has a power dissipation of 9W. It can support large RNN models. However, this accelerator uses reduced precision (i.e., 5 bits) to store the weights and to perform computations. Consequently, the model accuracy decreases by 7%, which is not acceptable for speech recognition applications. In this accelerator, accessing the main memory is avoided by storing the complete model in local on-chip memory. However, this solution does not scale up since it can only store models of up to 2 MiB in size, which is too small for complex speech recognition models. For instance, the EESEN [69] model cannot be store complete on-chip on this accelerator since it requires 5 MiB. In this thesis, we aim to support RNN applications from a variety of domains without degrading the base accuracy of the

[2] Computational units that compose an RNN layer. Formally presented in Chapter 2.

[3] Qm.n is a fixed-point format to represent decimal numbers. The m represents the bit-width of the integer part, whereas the n represents the bit-width of the fractional part.

models.

BrainWave [29] is a highly optimized accelerator that focuses on high performance and high throughput. It is composed of a vast array of matrix-vector multipliers and multi-functional units used to compute activation functions. One of the significant issues with this proposal is that the resource utilization is only 3.5% when evaluating small batches (i.e., one input sequence) [29]. Furthermore, its peak power computation is 125W, which is not amenable to edge devices since BrainWave is designed for data-centers.

ASICs

The Tensor Processing Unit (TPU) [51] is a state-of-the-art accelerator for deep neural networks. It is composed of a systolic array of processing elements (PEs) and an on-chip memory for weights and activations. TPU is mainly tailored to CNN, but it could also be employed with RNNs. Usually, an output stationary dataflow is employed to evaluate RNN models on TPU [81, 87]. Nonetheless, the resource utilization of TPU, when used for RNN inference, is only 18%. Furthermore, TPU has a power consumption of 40 watts [51], and a more recent version has a power dissipation of 200 watts [37]. TPU is designed for high performance and servers, whereas our work is tailored to high energy efficiency.

Edge TPU [38] is a low power version of TPU, which dissipates 0.5 watts for each TOPS. It is capable of performing 4 TOPS (tera-operations per second). Furthermore, it uses 8-bit fixed-point arithmetic. Finally, Edge TPU is tailored to CNNs and fully-connected networks, whereas this work focuses on RNNs.

The LSTM-Sharp [112] is another high-performance RNN accelerator that focuses on increasing resource utilization. The primary problem addressed in the LSTM-Sharp is the extra padding that occurs when performing a matrix-vector multiplication and the number of multipliers per tile are not multiples of the input vector length. This issue is more apparent in accelerators that employ a large number of multipliers such as BrainWave [29]. However, in this thesis we target the low power segment which usually uses a small number of multipliers, largely reducing this problem.

1.3.2 Energy-Saving Techniques

Deep Neural networks usually have much redundancy [45], which is exploited by techniques such as pruning to compress the DNN models. More specifically, pruning is a technique that removes the weights that are not contributing to the prediction hence reducing the model size. However, since many weights are pruned, the model becomes sparse, and special care must be taken when performing the matrix-vector multiplications. In this regard, several works employing pruning to increase the energy efficiency of RNN inference have been proposed [43, 42].

ESE [43] is an RNN accelerator which is explicitly designed for a speech recognition network, TIMIT [34]. ESE achieves high energy-efficiency by pruning 90% of the model. Also, weights are stored using 12-bits fixed-point. Furthermore, the original accuracy of the model is preserved by pruning and retraining. On the contrary, in this thesis, we aim to support a variety of application

domains. Moreover, we focus on improving energy efficiency by increasing weight reuse and reducing computations at run-time, whereas pruning is performed offline. Finally, our solutions are orthogonal to the techniques presented in [43].

MASR is another ASIC based RNN accelerator which employs pruning to improve energy-efficiency [42]. This accelerator is designed to exploit the sparsity in the weights and activations. Furthermore, it avoids storing and computing null values. Moreover, it presents a co-designed technique to encode the sparse weights and activations. The principal focus of this accelerator is sparse models, whereas this thesis targets dense models.

Wang et al. [107] proposes an FPGA accelerator that employs block-circulant matrices to compress the weights. By applying this technique, compression ratios of 16x can be obtained. However, the accuracy of the model is degraded. On the contrary, in this thesis we strive to propose solutions that do not degrade the original accuracy of the models.

1.4 Thesis Organization

The organization for the rest of this thesis is as follows.

Chapter 2 provides detailed information about RNNs. Moreover, we explain the principal RNN topologies: Long Short Term Memory and Gated Recurrent Unit.

Chapter 3 presents the evaluation methodology. First, we describe the infrastructure employed to estimate energy consumption, performance, power, and area of our proposals. Second, we present the software tools used to implement and test our techniques. Finally, we describe the set of RNN workloads used in our proposals.

Chapter 4 describes E-PUR, our proposal for low power RNN inference. First, we evaluate and benchmark several RNN applications to identify the main bottlenecks and sources of energy consumption. Second, we propose a highly optimized accelerator tailored to RNN inference. Finally, we describe our technique to maximize the temporal locality of the weights and improve their reuse.

Chapter 5 presents our technique for computation reuse. We first analyze several RNN application to find the potential for reuse. Then, we present a novel predictor for reusability and the principal motivation behind its choice. Finally, we describe Neuron Level Fuzzy Memoization for RNNs, a technique that improves energy efficiency by reusing previous computations.

Chapter 6 proposes a technique for selecting and employing lower precision dynamically. In this proposal, we analyze the impact of the precision used on the RNN state and the RNN's accuracy. Then, we use the RNN state to identify which computations will be done using a lower precision. Finally, we describe our scheme which employs a simple, yet effective hardware solution.

Chapter 7 presents a novel energy-efficient batching scheme for RNNs. Firstly, we identify the major inefficiencies in the state-of-the-art batching solutions for RNNs. Secondly, we describe our proposal to address the major issues of performing RNN batching efficiently. Finally, we show results for our schemes on top of E-PUR and TPU.

Chapter 8 outlines some of the future steps and open research areas. Finally, we summarize the main conclusions of this thesis.

2

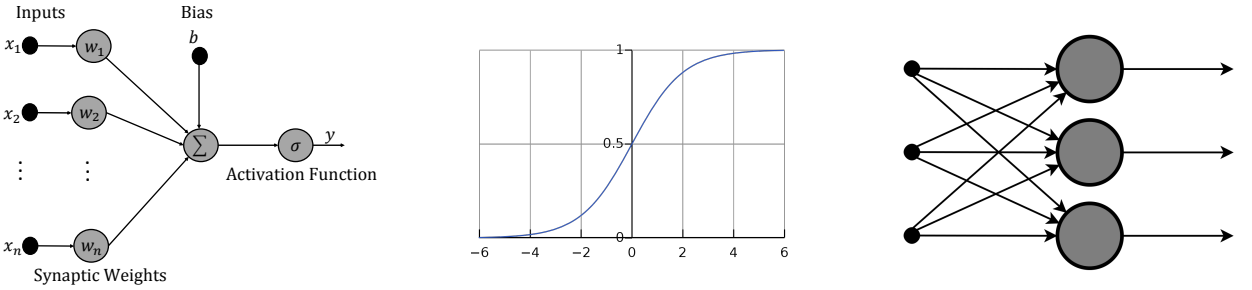
Background on Recurrent Neural Networks

In this chapter, we provide background information on Recurrent Neural Networks. First, we present a brief overview of Artificial Neural Networks and their building blocks. Second, we briefly describe the feed-forward neural networks. Then, we offer detailed information on RNNs and their different topologies. Finally, we discuss several topics and techniques that are employed throughout this work.

2.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are networks of simple processing units (i.e., neurons) whose initial design was motivated by the fact that the brain processes information in an utterly different manner than conventional digital computers [46]. However, nowadays, their design is far from their initial bio-inspired design [98]. Contrary to digital computers, the brain can learn and adapt [88]. Also, the brain can perform specific tasks such as pattern recognition and perception faster and more efficiently (i.e., energy-wise) than digital computers [46]. In theory, an artificial neural network can perform any arbitrary mapping from one vector space to another [68]. For the sake of simplicity, we refer to artificial neural networks as neural networks.

Figure 2.1c shows a high-level view of a neural network; its working principle is as follows. First, the input information is received and processed by the neurons. Then, the neurons send out the processed data to the network output and/or other neurons. The relevance of the information received by a neuron is determined by its connections (i.e., synaptic weights) with the network input and/or with other neurons. Neural networks are trained for a specific task by adjusting the weights until a well-defined objective function is fulfilled, and usually, a learning algorithm such as back-propagation [47] is employed. Accordingly, the learned knowledge is stored in the connections



(a) Artifical Neuron. (b) Sigmoid Function. (c) Artificial Neural Network.

Figure 2.1: Artificial Neural Networks. Its main component is the *neuron* (a). A sigmoid function is commonly used as an activation function (b). An artificial neural network (c) is created combining several neurons, and thus it can be trained to identify complex patterns.

of each neuron. A formal definition of an artificial neural network is provided in [46]: *"An Artificial Neural Network is a massively parallel distributed processor made up of simple processing units that has a natural propensity for storing experiential knowledge and making it available for use".*

2.1.1 Artificial Neurons

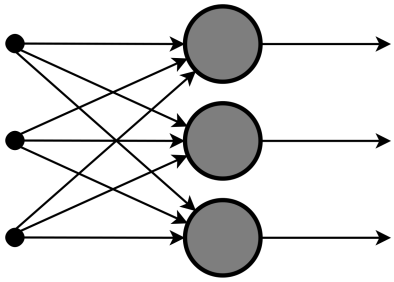
A *Neuron* is the basic computation unit of neural networks, shown in Figure 2.1a. As mentioned before, neurons are connected to external inputs or the outputs of other neurons via the synaptic weights. Then, the information received by the neurons is filtered out by the synaptic weights, as can be seen in Figure 2.1a. A neuron produces an output by summing all the filtered data acquired through its connections, and then applying an *activation function* (i.e., a sigmoid function, shown in Figure 2.1b). For the rest of the thesis, we refer to the synaptic weights as weights for the sake of simplicity.

Formally, the computations carried out by a neuron are shown in Equation 2.1. First, a linear combination of the neuron's input ($\vec{x}$) and the weights ($\vec{w}$) is performed. Also, the bias (b) is added. Then, the output of the neuron (y) is computed using a nonlinear function (σ). More specifically, the primary computation performed by a neuron is an inner product (i.e., dot product) between $\vec{x}$ and $\vec{w}$. Note that the bias term can be considered as an input with a weight whose value is one. Therefore, for the sake of simplicity, we omit the bias for the rest of the thesis.

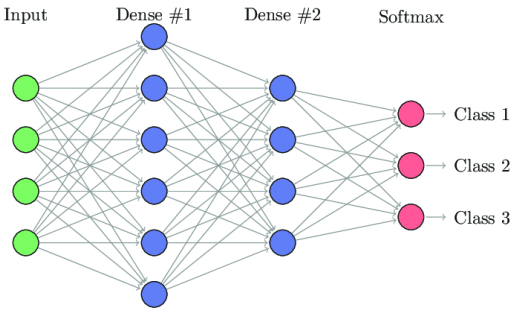
$$y = \sigma(\sum \vec{w}\vec{x} + b) \tag{2.1}$$

2.1.2 Feed-forward Neural Networks

Neural Networks can be classified into two categories: Feed-forward Multi-Layer Perceptrons (MLPs) and Recurrent Neural Networks (RNNs). MLPs do not have feedback from their output towards their inputs. On the contrary, RNNs include feedback loops. Although in this work we



(a) Single-layer Feed-forward Neural Network.



(b) Multi-layer Perceptron (MLP).

Figure 2.2: MLPs are composed of millions of neurons arranged by layers. The layers between the input layer and the output layer are called hidden layers. All the layers in both figures are fully-connected.

focus on RNNs, feed-forward neural networks are presented in this section since they are an essential building block of RNNs.

A feed-forward MLP is composed of many neurons organized in different layers [68]. Figure 2.2a shows a feed-forward neural network with only one layer, a.k.a. perceptron, whereas an MLP is shown in Figure 2.2b. The layers between the input and the output layer are called *hidden layers*. Note that the input layer does not perform any computation, and thus it is not included when counting the number of layers in a neural network. For instance, the number of layers of the neural network shown in Figure 2.2b is three: the output layer (*Softmax*) and two hidden layers (*Dense #1* and *Dense #2*).

The current trend in MLPs is to create very deep networks with many layers to increase recognition accuracy. The reason is that neural networks with a large number of neurons (i.e., a wide network) and networks with a small number of hidden layers (i.e., a shallow network) are excellent at memorization. Consequently, they cannot generalize, whereas deep neural networks can capture richer structures [26].

Generally, the layers of a neural network where every neuron in the layer is connected to the output of every neuron in the previous layer is called a fully-connected layer. Moreover, neural networks where all the layers are fully-connected are called fully-connected networks [88]. Otherwise, they are called *partially- connected* networks. Formally, the computations carried out by a fully-connected layer are shown in Equation 2.2. They are defined as a matrix-vector multiplication between W and $\vec{x}$. The rows of the weight matrix W contain the connections for each neuron in the layer, whereas $\vec{x}$ is the output vector of the previous layer (i.e., one element per neuron of the previous layer). Note that W for fully-connected layers is a dense matrix, whereas it is a sparse matrix for partially-connected layers. As with neurons, for the sake of simplicity, we omit the bias

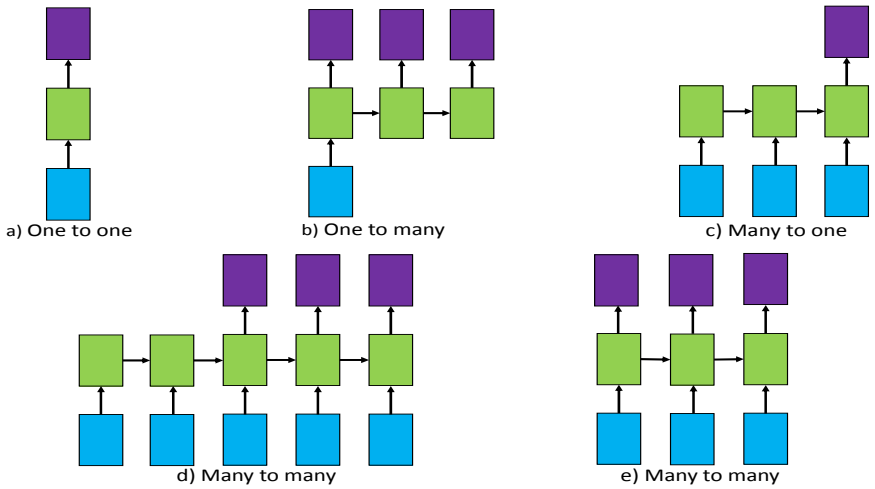


Figure 2.3: Modeling sequence to sequence problems with RNNs. In each sub-figure, blue rectangles correspond to inputs, violet rectangles to outputs, and green boxes to the RNN cell. (a) This is the typical case solved by standard feed-forward networks. (b) Sentiment analysis and video classification are tasks in which a sequence is mapped to one output vector. (c) Applied to problems such as image captioning and text generation where the input is a vector. (d) This configuration has been used for machine translation tasks in which the input and output sequences may have varying and different lengths. e) This architecture has been used for predicting at each step the following character in a sentence.

vector for the rest of the thesis.

$$\vec{y} = \sigma(W\vec{x} + \vec{b}) \tag{2.2}$$

MLPs are trained by adjusting the weight matrix of each layer. Typically, a variation of the back-propagation algorithm, such as ADAM [57] or RMSprop [84], is employed during training. In this work, we focus on inference; thus, we do not detail the training process of neural networks.

2.2 Recurrent Neural Networks

RNNs provide a start-of-the-art solution for sequence to sequence problems. Unlike feed-forward neural networks, RNNs include loops, which allows them to use a vast amount of context information during inference. In this section, we provide background information on RNN and the motivation behind its different topologies. First, we present a brief description of sequence to sequence problems. Second, the vanilla RNN and deep RNNs are described. Then, we detail the Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) architectures. Finally, we review Bi-directional RNNs.

2.2.1 Sequence to Sequence Problems

Sequence to Sequence problems involve mapping an input sequence (e.g., audio frames, a sentence) of one or more elements to another sequence, which can also have one or more elements [66, 97]. RNNs are suitable for this kind of problem because they do not require an input or output sequence with fixed length [66]. Also, due to the recurrent connections, they can learn short and long temporal and spatial dependencies on the input data [97].

Figure 2.3 shows several sequence to sequence models. Sentiment analysis and video classification can be modeled with Figure 2.3b, whereas Image captioning could be represented as in Figure 2.3c. Moreover, problems where the input and output sequence have a variable length are modeled as in Figure 2.3d. On the contrary, problems such as language modeling (i.e., the input and output sequence have the same length) are modeled as in Figure 2.3e. Note that Figure 2.3a is the traditional application where feed-forward neural networks are used.

2.2.2 Vanilla RNN Cell and Deep RNNs

Vanilla RNN Cell

Figure 2.4a shows the most basic architecture of an RNN, commonly referred as Vanilla RNN cell [109, 97]. The input (i.e., X) to a vanilla RNN cell is a sequence of real value vectors, i.e., $[\vec{x}_1, \vec{x}_2, ...\vec{x}_N]$ where N is the number of elements in the input sequence. Similarly, the output (i.e., Y) is also a sequence of real value vectors, i.e. $[\vec{y}_1, \vec{y}_2, ...\vec{y}_M]$ where M is its length. Note that N is not necessarily equal to M. Conventionally, an element of the input sequence (i.e., $\vec{x}_t$) is called a *time-step* [66].

As Figure 2.4a shows, a vanilla RNN cell (RNN cell for simplicity) includes forward connections and recurrent connections (also called feedback connections). The forward connections operate on $\vec{x}_t$ (i.e., the current time-step of the input sequence). On the other hand, the recurrent connections work on $\vec{y}_{t-1}$ (i.e., the output of the RNN cell on the previous time-step). Figure 2.4b shows an unrolled RNN cell which is evaluated sequentially for each time-step of the input sequence, i.e., (x_1) to (x_N). In the unrolled view of the network, recurrent connections are shown as horizontal connections. Note that these horizontal connections correspond in fact to connections from the output of one cell to the input of the same cell.

Conceptually, the forward and recurrent connections can be defined as two independent fully-connected layers (without the activation function) whose weight matrices are W and U, respectively. Also, note that W and U always have an identical number of rows and columns. Formally, the output of an RNN cell is computed according to Equation 2.3. As seen in Equation 2.3, to compute the output of an RNN cell for a time-step $\vec{x}_t$, two matrix-vector multiplications are done: one between the forward connections W and the current time-step $\vec{x}_t$, and the other between the recurrent connections U and the previous output $\vec{y}_{t-1}$. Then, the results of these two multiplications are added together. Finally, to compute the output $\vec{y}_t$, the result of this summation ($\vec{h}_t$) is fed to an activation function which is usually a *hyperbolic tangent* (ϕ in Equation 2.3) for Vanilla RNN cells.

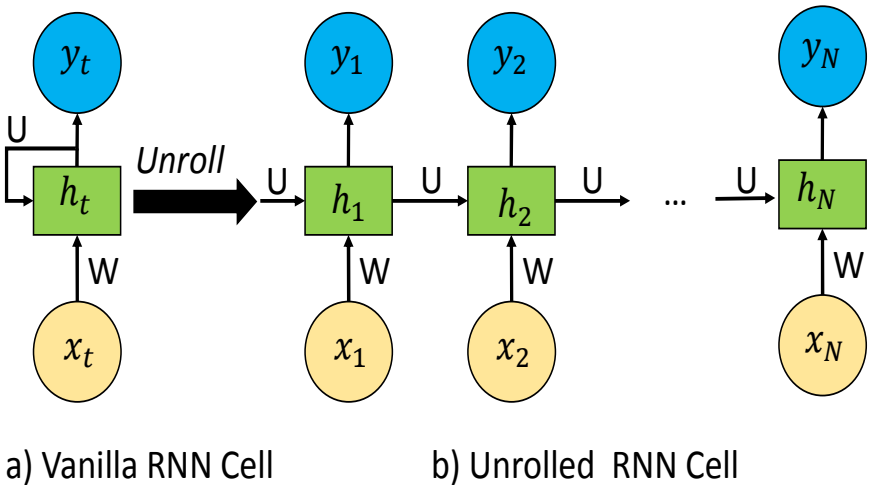


Figure 2.4: Vanilla RNN cell. Its output is a function of the current time-step x_t and the previous output y_{t-1}. The figure in the right shows the cell unrolled for an input sequence $X = [x_1, x_2, .., x_N]$.

$$\begin{aligned} \vec{h}_t &= W\vec{x}_t + U\vec{y}_{t-1} + \vec{b} \\ \vec{y}_t &= \phi(\vec{h}_t) \end{aligned} \tag{2.3}$$

Regarding neurons, note that the activation function is applied to $\vec{h}_t$. Hence, conceptually, an element (n_k) of the vector $\vec{y}_t$ can be viewed as the output of a neuron since it follows to the definition of neuron given in 2.1.1. Therefore, for RNNs, the output of neuron n_k is computed by adding the result of the inner product between row $\vec{w}_k$ (i.e., forward connections) and the current time-step $\vec{x}_t$, to the result of the inner product between row $\vec{u}_k$ (i.e., recurrent connections) and the previous output $\vec{y}_{t-1}$. Also, consider that when stating the number of neurons in an RNN, it refers to the dimensionality of $\vec{y}_t$. Thus, it is equal to the number of rows in the matrix containing the recurrent or forward connections. Finally, some works used the term *units* instead of neurons. However, throughout this thesis we use the term neuron unless it is explicitly stated otherwise. Finally, since RNN computations are mostly done using vectors and matrices, we refer to a given vector $(\vec{x})$ just as (x) for the sake of readability.

Regarding the input sequences, since RNN cells are employed in a plethora of applications from different domains, the elements of an input sequence must be encoded as a real-value vector (e.g., a hot vector). Moreover, for some applications such as sentiment analysis [23], the input sequences do not have an explicit temporal component (i.e., words in a sentence); nonetheless, the convention is to call every single element on an input sequence a *time-step*. We follow this convention in this thesis.

Deep RNN

Deep RNNs are created by stacking together several RNN cells in a layered manner. In the context of deep RNNs a layer consists of one or more RNN cells. Similar to deep feed-forward

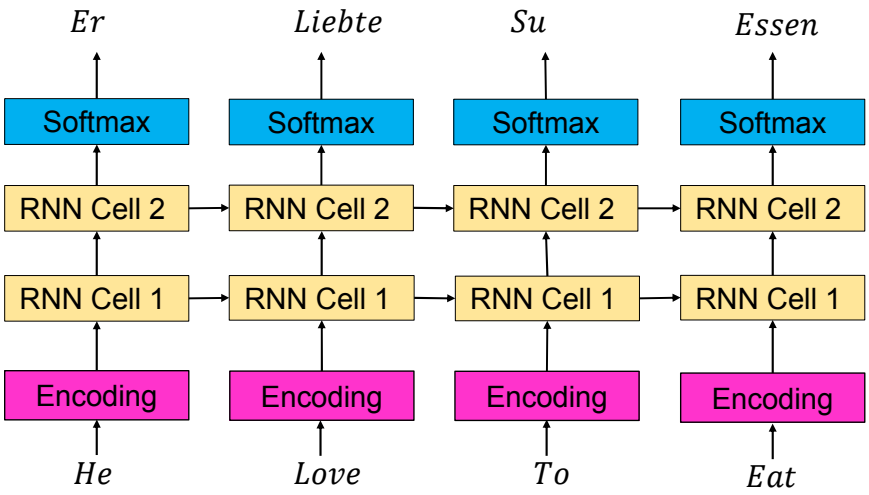


Figure 2.5: Deep RNN employed to translate an English sentence to German. First, words are encoded into a real value vector. Then, they are processed sequentially for each RNN cell.

neural networks, deep RNNs include an input layer, one or more hidden layers, and an output layer. Figure 2.5 shows an unrolled Deep RNN with two hidden layers. In this logical view of the network, recurrent and forward connections are shown as horizontal and vertical links, respectively. Therefore, the output of a given RNN cell is used as input to the next RNN cell in the same time-step (forward link), but also as input to the same RNN cell in the next time-step (recurrent link).

As an example, Figure 2.5 shows the process of translating an input sentence from English to German using a deep RNN. As it can be seen, each word in the input sentence is first encoded as a vector (e.g., $\vec{x}_t$, a one-hot vector). Then, each of the encoded-words (i.e., a time-step) is passed to the first RNN layer, which evaluates them sequentially from x_1 to x_4. Note that the output of evaluating a time-step is fed to the upper layer and also used when processing the next input word. Once the first layer is evaluated for all the time-steps, the same process is followed for the rest of the hidden layers. Finally, the output of the last RNN cell is given to a Softmax function that generates a probability distribution of the most likely words. Finally, output word for each time-step is selected using either a greedy approach or a beam search [110].

As mentioned in 2.1.2, the primary motivation for deep neural networks is that they are better at generalizing than shallow networks [26]. For instance, consider the machine translation model [110], which translate sentences from English to German. When the model is configured with four RNN cells and 1024 neurons, its accuracy is 19 Bleu [74][1] . However, increasing the number of RNN cells to 8 raises the model accuracy to 26 Bleu. On the contrary, doubling the number of neurons does not increase the model accuracy.

Vanilla RNNs can capture and exploit short term dependencies in the input sequence. However, capturing long term dependencies is challenging since useful information tends to dilute over time [15, 48]. To exploit long term dependencies, Long Short Term Memory (LSTM) [48] and Gated Recurrent Units (GRU) [20] networks were proposed. These types of RNNs represent the

[1]It is a metric for evaluating a generated sentence to a reference sentence. It works by counting matching n-grams in the candidate translation to n-grams in the reference text. The larger the Bleu, the better.

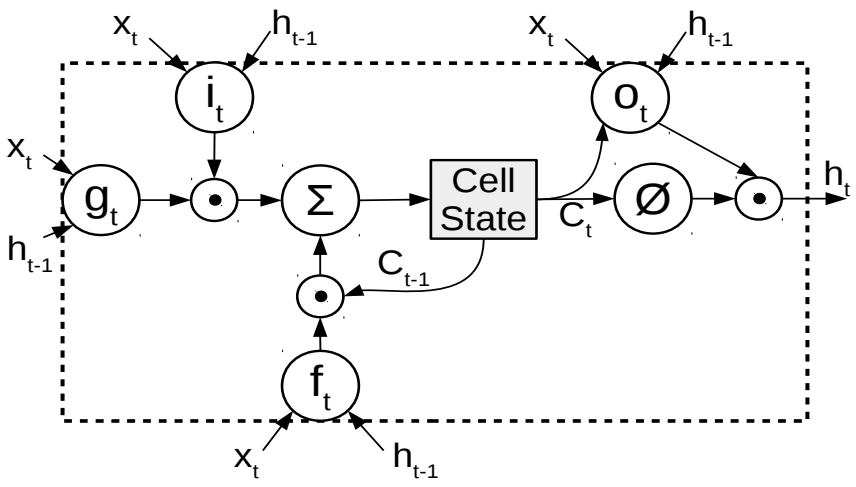


Figure 2.6: Structure of an LSTM cell. $\odot$ denotes an element-wise multiplication of two vectors. ϕ denotes the hyperbolic tangent.

most successful and widely used RNN architectures. In the next subsections, we describe their architecture and formal model.

2.2.3 Long Short Term Memory Cell

Figure 2.6 shows the structure of an LSTM cell. Contrary to a Vanilla RNN cell, an LSTM cell has a memory storage. Moreover, the flow of information going in or out of an LSTM cell is modulated. The critical component of an LSTM cell is the cell state, i.e. c_t, a vector that is stored in the cell memory. Moreover, an LSTM cell has four gates that are used to build the cell state and to control the output of the cell. In this context, a gate is a multiplicative node which is composed of two independent fully-connected layers operating on the forward and recurrent connections, respectively.

The gates in an LSTM cell control how the information is added or removed from the cell state. More specifically, for an input sequence (X), the cell state is updated by three of the gates in the following manner. First, the updater gate (g_t, whose computations are shown in Equation 2.4c) modulates the amount of input information that is considered a candidate to update the cell state. Then, the input gate, (i_t, Equation 2.4a) decides how much of the candidate information will be added to the cell state. On the other hand, the forget gate (f_t, Equation 2.4b) determines the amount of information to be erased from the previous cell state (c_{t-1}). Once these three gates are evaluated, the cell state is updated according to Equation 2.4d. Similarly, the output gate (o_t, Equation 2.4e) decides the amount of information that will be emitted from the cell to create the output (h_t).

$$i_t = \sigma(W_i x_t + U_i h_{t-1}) \tag{2.4a}$$

$$f_t = \sigma(W_f x_t + U_f h_{t-1}) \tag{2.4b}$$

$$g_t = \phi(W_g x_t + U_g h_{t-1}) \tag{2.4c}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{2.4d}$$

$$o_t = \sigma(W_o x_t + U_o h_{t-1}) \tag{2.4e}$$

$$h_t = o_t \odot \phi(c_t) \tag{2.4f}$$

The computations carried out by an LSTM cell are shown in Equations 2.4 (a)–(f). For these equations, $\odot$, ϕ, and σ denote element-wise multiplication, hyperbolic tangent, and sigmoid function, respectively. Note that the output of each gate is a vector. Conceptually each of the four gates is composed of multiple neurons. As mentioned before, each of them consists of two independent fully-connected networks, which are implemented as two matrix-vector multiplications. Therefore, to compute the output of neuron (n_k) in any of the gates, two dot-product operations are performed: one between the forward connections (w_k) and x_t and the other between the recurrent connections (u_k) and (h_{t-1}). Note that for LSTM cells, most of the computations requirements are due to the matrix-vector multiplications to compute the forward and recurrent connections. On the other hand, most of the memory requirements are due to the weight matrices W and U.

As it can be seen in Equation 2.4d, the cell state (c_t) is a vector. Conceptually, the value for a given element (c_k) of the cell state is updated based on the value of neurons i_k, f_k, and g_k from the input, forget, and updater gates respectively. Moreover, the cell output (h_t) is also a vector. Hence, the value of the element h_k of the cell output is computed using the value of neuron o_k from the output gate and element c_k of the cell state.

Multiple LSTM cells can be stacked in multiple layers to create a deep LSTM network. Deep LSTM networks are very similar to the deep RNNs shown in 2.2.2. However, they employ LSTM cells instead of Vanilla cells.

2.2.4 Gated Recurrent Unit

GRUs are a class of RNNs motivated by the LSTMs, but they are much simpler to compute and implement. Analogous to an LSTM cell, a GRU cell includes gates to control the flow of information inside the cell. However, GRU cells do not have an independent memory cell (i.e., cell state). On the contrary, they only include a hidden state (i.e., h). Note that GRUs do not include an output gate and, hence, the entire hidden state is emitted from a GRU cell at each time-step.

As it can be seen in Figure 2.7, in a GRU cell the update gate (z_t, Equation 2.5a) controls how much of the new hidden state ($\widetilde{h}_t$, Equation 2.5c) is used to update the cell hidden state (h_t, Equation 2.5d). On the other hand, the reset gate (r_t, Equation 2.5b) modulates the amount of information that is removed from the previous hidden state.

$$z_t = \sigma(W_z x_t + U_z h_{t-1}) \tag{2.5a}$$

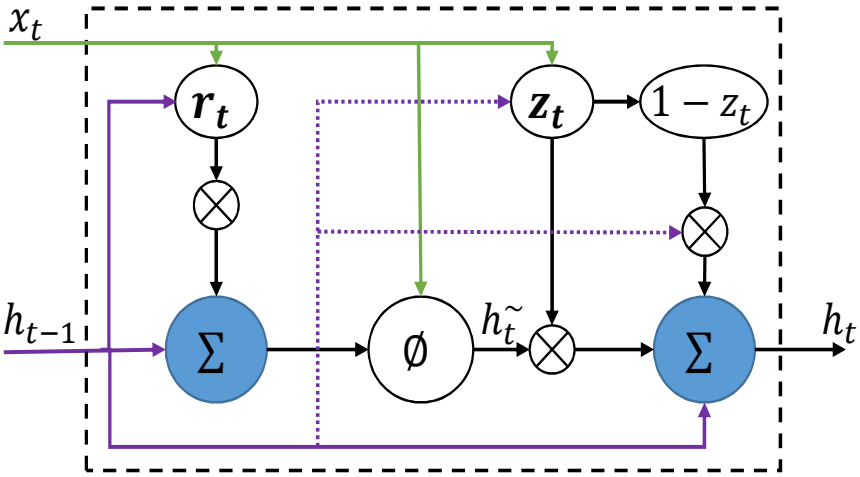


Figure 2.7: Structure of a GRU cell.

$$r_t = \sigma(W_r x_t + U_r h_{t-1}) \tag{2.5b}$$

$$\widetilde{h}_t = \phi(W_h x_t + r * U_h h_{t-1}) \tag{2.5c}$$

$$h_t = z * h_{t-1} + (1 - z) * \widetilde{h}_t \tag{2.5d}$$

The computations carried out by a GRU cell are shown in Equations 2.5 (a)–(d). Note that they are very similar to the equations for an LSTM cell (Equations 2.4). Analogous to LSTM cells, the gates in a GRU cell are composed of multiple neurons, and they consist of two independent fully-connected networks, which are implemented as two matrix-vector multiplications. Also, neurons are defined as in 2.2.3.

Regarding deep GRU networks, they are essentially deep LSTM networks but using GRU cells. For the rest of the thesis, we refer to GRU and LSTM cells as RNN cells. Moreover, we use the term cell state to refer to the hidden state (h_t) of GRU cells and the cell state (c_t) of LSTM cells. Finally, most of the applications using RNNs employ either LSTM or GRU cells; we do not focus on vanilla RNN cells.

2.2.5 Bi-directional RNNs

Layers in a deep RNN can be unidirectional or bidirectional. Unidirectional layers only use past information to perform inference during the evaluation of a time-step. On the other hand, bidirectional layers exploit both past and future context information and, typically, they provide higher accuracy. Therefore, Deep Bidirectional RNNs (BiRNN) deliver state-of-the-art accuracy for multiple sequence processing problems [14, 40, 89].

Figure 2.8 shows an unrolled BiRNN network with one hidden layer. The bidirectional layers consist of two RNN cells; the first one processes the information in the forward direction, i.e. (x_1)

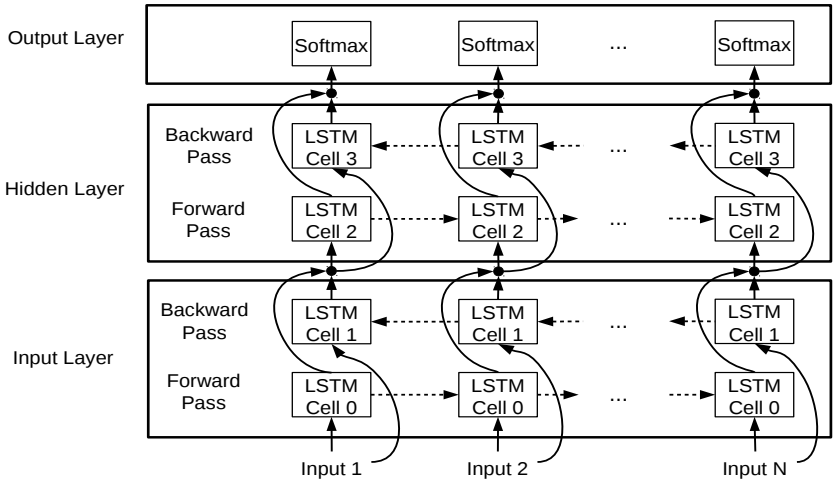


Figure 2.8: BiRNN.

to (x_N), while the second one processes the input sequence in the backward direction, i.e. (x_N) to (x_1). Figure 2.8 shows multiple instances of these two cells for each layer, which correspond to numerous recurrent uses of the same two cells, one for each time-step in the input sequence. In the unrolled view of the network, recurrent connections are shown as horizontal connections, either left-to-right or vice versa, and they correspond in fact to connections from the output of one cell to the input of the same cell. In a given layer, the outputs of the LSTM cells in both forward and backward directions are concatenated to create the input (x_t) for the next layer. Finally, a BiLSTM network includes a Softmax output layer that produces the final output of the network. For example, for speech or text applications, the outputs from the Softmax represent the likelihoods of the different characters, phonemes, or words at each time-step.

2.3 Additional Background

2.3.1 Linear Quantization

Linear quantization is a technique to reduce memory footprint and computational cost that is used extensively on DNNs [110, 51]. The main idea consists of approximating a full precision value (y) using an approximate value (y_q) which is expressed as function of an integer index and a quantization step (q) as shown in Equations 2.6 (a)–(c). For these equations, n is the bit-width of the integer index i_k, e.g., 8 bits, and α is the maximum absolute value of y.

$$q = \frac{\alpha}{2^{n-1}} \tag{2.6a}$$

$$i_k = round(y/q) \tag{2.6b}$$

$$y_q = q * i \tag{2.6c}$$

Regarding an RNN cell, once the weights and inputs have been quantized, the inner product to compute the output of a neuron is done using integer arithmetic (Equation 2.7a). Then, the result is converted back to a floating-point (Equation 2.7b) value and used to compute the activation function (Equation 2.7c).

In Equations 2.7 (a)–(c) w_i and x_i are the quantized index for each element of the weight and input vector. Moreover, q_x and q_w are the quantization steps for the weights and inputs, respectively. Multiplications are performed using 8 bits multipliers while summations and accumulations are normally performed with a higher number of bits (e.g., 24 bits). Furthermore, the activation functions for each neuron is computed using floating numbers (i.e., fp16 or fp32).

$$z_i = \sum w_i * x_i \tag{2.7a}$$

$$z_k = z_i * q_x * q_w \tag{2.7b}$$

$$y_q = \sigma(z_k) \tag{2.7c}$$

2.3.2 Binarized Neural Networks

As mentioned earlier, state-of-the-art DNNs typically consist of millions of parameters (a.k.a. weights) represented as floating-point numbers using 32 or 16 bits and, hence, their storage requirements are quite large. Linear quantization may be used to reduce memory footprint and improve performance [110, 51]. Moreover, real-time evaluation of DNNs requires a high energy cost. As an attempt to improve the energy-efficiency of DNNs, Binarized Neural Networks (BNNs) [22] or Bitwise Neural Networks [56] are a promising alternative to conventional DNNs. BNNs use one-bit weights and inputs that are constrained to +1 or -1. Typically, the binarization is done using Equation 2.8:

$$x^b = \begin{cases} +1 & \text{if } x >= 0, \\ -1 & \text{otherwise,} \end{cases} \tag{2.8}$$

where x is either a weight or an input and x^b is the binarized value which is stored as 0 or 1.

Regarding the output of a given neuron, its computation is analogous to conventional DNNs, but employing the binarized version of weights and inputs, as shown in Equation 2.9:

$$y_t^b = \sum w^b x_t^b \tag{2.9}$$

where w^b and x_t^b are the binarized weight and input vectors respectively. Note that evaluating the neuron output (y_t^b) only involves multiplications and additions that, with binarized operands, can be computed with XNORs and integer adders. BNN evaluation is orders of magnitude more efficient, in terms of both performance and energy, than conventional DNNs [22]. Nonetheless, DNNs and RNNs still deliver significantly higher accuracy than BNNs [80].

2.3.3 Batching

Sequence batching (we will also refer to it as batching for short) is a well-known technique that is commonly used to increase throughput. Batching is used during DNN training almost all the time. In this context, a batch is a set of inputs (i.e., input sequences for RNNs or images for a CNN). The number of batched inputs is usually called the *batch size*. Batching is also employed during inference by machine learning systems handling multiple requests (e.g., data centers providing service to many users). Batching together multiple requests increases parallelism and improves weight reuse, i.e. weights can be fetched once from main memory and used to evaluate multiple sequences. We present a detailed discussion of batching techniques for RNNs in Chapter 7.

3

Experimental Methodology

In this chapter, we detail the methodology that is followed during this work. First, we present the simulator and tools that we employ to simulate and implement our proposals. Then, we describe our benchmarks and the methodology that we follow to evaluate the impact of our proposals in terms of accuracy.

3.1 Hardware Simulation Infrastructure

The standard methodology employed to model hardware architectures tailored to RNN acceleration involves using cycle-level simulators to estimate timing and activity factors. Moreover, pipeline components are implemented in Verilog and synthesized for a given technology to estimate their energy consumption, timing and area. Finally, the total execution time, energy consumption and power of the accelerator are estimated by combining the results from the cycle-level simulator and the synthesis process. In this work, we model two RNN accelerators: E-PUR and TPU.

3.1.1 E-PUR

To model E-PUR, we implemented a cycle-level simulator in Python. This simulator estimates the execution time of an RNN model running on top of it by accurately modeling each of the hardware components of E-PUR's pipeline. At each execution cycle, the dynamic activity of each pipeline component is calculated and recorded. Note that the behavior and memory access pattern of RNNs is very predictable, and thus many pipeline components are simulated using an analytical model. After a simulation is completed, the simulator provides the total execution time and overall energy consumption (dynamic and static) of the accelerator. Furthermore, it provides the power

Table 3.1: Hardware parameters for E-PUR.

Parameter	E-PUR
Technology	32 nm
Frequency	500 MHz
Intermediate Memory	1.5 MB
Weights Memory	2 MB per CU
Inputs Memory	8 KB per CU
DPU Width	16 operations
MU Operations	cycles: 2 (ADD), 4 (MUL), 5 (EXP)
MU Communication	2 cycles
Peak Bandwidth	30 GB/s

dissipation and the dynamic activity of each pipeline component. Note that the simulator only estimates the execution cycles and the activity factors. To determine timing and energy consumption, it combines the energy and timing estimations obtained using the following tools.

Regarding the latency, area and power dissipation of each of the individual pipeline components, we implement them in hardware by creating their RTL description in Verilog. Then, we use Synopsys Design Compiler [99] to synthesize each pipeline component employing a 32 nm technology library. The compiler reports the power dissipation (static and dynamic) and the critical-path delay. We use a typical process corner with a voltage of 0.78V, and an average switching activity is used to estimate dynamic power. Moreover, to estimate the energy consumption and delay of the memory components (i.e., buffers and scratch-pad memories) of E-PUR, we use CACTI selecting energy-delay as the optimization function. Finally, to estimate the timing and energy consumption of the main memory we employ MICRON's power model [70]. We model 4 GB of LPDDR4 DRAM.

Regarding the clock frequency, we use the delays reported by Synopsys Design Compiler and CACTI [71] to set the frequency such that most hardware structures operate on a single clock cycle. We also evaluate alternative frequency values to minimize energy consumption.

Table 3.1 shows the standard hardware configuration of E-PUR. However, note that in some of our proposals, we use a slightly different configuration as a baseline. For each of our proposals, we detail the modifications done to the baseline in the corresponding chapter.

3.1.2 TPU-like architecture

To model a TPU-like architecture, we use the SCALE-Sim [87, 86] simulator with the configuration parameters shown in Table 3.2. We employed an output stationary dataflow [87, 19]. The number of filters is set to the number of neurons in the model. Also, the input features are set to the number of weights per neuron. On the other hand, the width and height of the filters are set to one, whereas the width of the input features is set to the batch size. Pipeline components are modeled using the methodology that was described for E-PUR. Also, the energy consumption and timing is estimated in a similar manner.

Table 3.2: Configuration of TPU-like accelerator.

Parameter	Value
Frequency	700 MHz
SRAM Buffer	24 MiB
Systolic Array PEs	128x128
Dataflow	Output Stationary

Table 3.3: Tegra X1 parameters.

Parameter	Value
CPU	4-core ARM A-57
GPU	256-core Maxwell GPU
Streaming Multiprocessors	2 (2048 threads/proc)
Technology	20 nm
Frequency	1.0 GHz
CPU L2 Cache	2 MB
GPU L2 Cache	256 KB
Peak Bandwidth	25.6 GB/s

3.2 GPU Evaluation

For comparison purposes, we use an NVIDIA Tegra X1 SoC [72] whose parameters are shown in Table 3.3. Its energy consumption is measured by reading the registers of the Texas Instruments INA3221 power monitor included in the Jetson TX1 development board [72], as outlined in [30]. Furthermore, we only use the energy consumption of the GPU. Regarding the software implementation of the networks, we implemented them using high-level APIs (as explained in the next section). Then, to perform inference on the Jetson TX1, we rely on cuDNN [11], a GPU-accelerated and highly optimized library of primitives for DNNs.

3.3 Functional Evaluation and Benchmarks

The machine learning community offers a variety of software frameworks tailored to the implementation of deep learning models. Some software frameworks such as Keras [21] and PyTorch [79] have a very user-friendly API, whereas frameworks such as TensorFlow [2] work at a lower level of abstraction. All of them use the Python programming language to implement the models and a highly optimized back-end library, such as cuDNN [11] or cuBLAS [1], to perform computations. We use several of these frameworks in this work since our benchmark models are based on open source projects from different companies and research groups.

Our benchmarks include RNN models from different application domains. They have a different number of RNN layers and neurons. Moreover, they have either LSTM or GRU cells, and some of them include bidirectional layers. On the other hand, the length of the input sequence is also different for each RNN, and it ranges from 20 to a few thousand time-steps. Finally, we used the

test and train sets provided with each network during our experiments.

Regarding the functional evaluation of a model's performance, we employ the term accuracy to refer to how good or bad an RNN model is at predicting correctly the outputs. However, several metrics are used in the machine learning community to measure the performance of a model. Also, we use the term accuracy loss to refer to the fact that the accuracy of a model is worse than its baseline accuracy.

To evaluate the impact that our proposals may have on the accuracy of a given model, we implemented them on top of the original model. More specifically, for a given RNN model, we develop our techniques using its framework and compare the new accuracy against the baseline accuracy to measure potential accuracy loss.

In the next subsections we present a brief overview of each of our benchmark models and the metrics employ to determine their accuracy. Moreover, we show their characteristics in Table 3.4.

3.3.1 Accuracy Metrics of the Benchmarks

A variety of metrics are employed in the Machine Learning community to measure the accuracy of a DNN model: how good or bad a DNN model it is when performing a given task. In the next subsections, we describe the accuracy metrics used by the benchmarks evaluated in this thesis.

Top-N Accuracy

Top-N is an accuracy metric commonly employed in classification problems to measure the ratio of correct predictions over the total number of instances evaluated. Note that in classification problems, the predicted class is selected based on a probability distribution outputted by a Softmax function. In this case, the correct class could correspond to one of the top N predictions (i.e., the N classes with highest probabilities). Hence, for Top-N accuracy, a prediction is considered correct if it corresponds to one of the top N classes of the Softmax distribution. For instance, top-5 means that the right class corresponds to one of the five classes with highest probabilities outputted by the Softmax function.

Bilingual Evaluation Understudy (Bleu)

The Bleu [74] score is an accuracy metric typically used in problems that generate an output sentence such as machine translation and video captioning. Bleu compares a generated sentence to one or more reference sentences. The comparison is made by counting matching n-grams (usually up to 4-grams) in the generated sentence to n-grams in the reference sentences. Then, a weighted averaged of the matching n-grams is computed. The Bleu score is a value between 0 and 1, but it is generally multiplied by one hundred. A value of one indicates that the generated sentence matches precisely one of the reference sentences. In contrast, a score of zero shows no matching between the generated and the reference sentences. Therefore, a higher Bleu score is considered better.

Word error rate (WER)

Word error rate (WER) is a standard metric of the performance of a speech recognition system. It works by counting the number of insertions plus deletions plus substitutions required to convert the recognized word sequence into the reference word sequence, divided by the total number of words of the reference word sequence. Lower WER is considered better.

3.3.2 Benchmark RNN Models

IMDB Model

The IMDB Model [23] is employed to predict the user sentiment about a movie based on an input review. The final output of the model tells whether the user gave a positive or negative review. As dataset, it uses movie reviews from the IMDB movie database.

Regarding its parameters, it has 1 LSTM layer and 128 neurons. On the other hand, the input sequences have a fixed length of 80 time-steps. For this model, the accuracy metric used is top-1 accuracy. Its baseline accuracy is 86.5%. Finally, it is implemented in Python using the Keras library.

DeepSpeech2

DeepSpeech2 [7] is a speech recognition system for different languages. In our evaluations, we employ the Librispeech [73] dataset, which contains more than 900 hours of speech from audiobooks. Speech models for Librispeech are trained with a large vocabulary of 200K words. Since the input sequences to this model are usually composed of audio frames of 10ms, they tend to have a large length, which ranges from 100 to 2000 time-steps.

The DeepSpeech2 implementation used in our evaluations contains 5 GRU cells, and each one of them has 800 neurons. For this model, the baseline accuracy is 10.24 WER (Word Error Rate). Regarding the software implementation, we implemented it on Python and used the trained model (weights) from the original paper.

EESEN

EESEN [69] is another speech recognition system for the English language. It uses the Tedlium [83] dataset, that is collected from TED talks and consists of spontaneous speech in a noisy environment. It is trained with a vocabulary of 150K words. Similar to DeepSpeech2, the input sequences range from a few hundred to 2000 time-steps.

The EESEN model is composed of 5 BiLSTM layers, each one containing 320 neurons. In this model, the baseline accuracy is 23.8 WER. For the software implementation, we used the open-source model from the original paper, which is written in C++.

Table 3.4: RNN Networks used for the experiments.

Network	App Domain	Cell Type	Layers	Neurons	Accuracy	Dataset
IMDB Sentiment [23]	Sentiment Classification	LSTM	1	128	86.5% (Top-1)	IMDB dataset
DeepSpeech2 [7]	Speech Recognition	GRU	5	800	10.24 WER	LibriSpeech
EESEN [69]	Speech Recognition	BiLSTM	10	320	23.8 WER	Tedlium V1
GNMT [110]	Machine Translation	LSTM	8	1024	29.8 Bleu	WMT'15 En→Ge
SHOW TELL [104]	Image Description	LSTM	3	512	32.2 Bleu	MSCOCO

Google Machine Translation (GNMT)

GNMT [110] is an RNN model for machine translation proposed by Google. It is based on seq2seq [97] architecture, which is composed of an encoder and a decoder. Both the decoder and decoder consist of BiLSTM and LSTM layers. In total, this model has 10 LSTM layers and 1024 neurons per layer. Our evaluations are based on the English to German dataset (WMT'15 En→Ge) [16].

For this model, the input sequences are sentences, and their number of time-steps ranges from 20 to 100. Moreover, this model is implemented in TensorFlow. Finally, it has a baseline accuracy of 29.8 Bleu.

Showtell

Showtell [104] is an RNN model for image description, i.e. it generates a textual description of the action in an image. It is composed of 3 LSTM layers, each one containing 512 neurons. It is implemented in TensorFlow and employs the MSCOCO [64] dataset, which includes thousands of images. The input sequences in these models range from 20 to 100 time-steps. Finally, it has an accuracy of 32.2 Bleu.

4

Energy-Efficient Processing Unit

In this chapter, we describe E-PUR, an energy-efficient processing unit tailored to RNN inference. E-PUR is our proposal to overcome the problem of supporting large RNN models while providing real-time performance with an energy consumption amenable for low power devices. First, we describe our approach to evaluate RNNs on a custom architecture. Second, we detail the overall architecture of E-PUR and its pipeline. Third, we describe our technique to maximize weight locality. Finally, we present the experimental results and conclusions.

4.1 RNN inference on E-PUR

State-of-the-art hardware implementations [41, 43, 62, 61] for RNNs rely on storing the synaptic weights on-chip in order to avoid expensive off-chip memory accesses. Commonly, there are two possible approaches to compute the output of an RNN given an input sequence ($X = [x_1, x_2, .., x_N]$): vertical or horizontal computations. Vertical computations refer to the case where the final output of the RNN for time-step (x_t) is computed before proceeding with the next time-steps. More specifically, time-step (x_t) is evaluated for all the RNN layers, and then the evaluation proceeds to time-step (x_{t+1}). One drawback of this approach is that, unless the entire RNN model is kept locally in on-chip memory, the temporal locality of the weights is severely affected since the entire weight matrices for the forward and recurrent connections of each RNN layer must be fetched from main memory for each time-step of the input sequence. Therefore, this approach is inefficient energy-wise. Another drawback is that it only works for unidirectional RNN layers, since for a bi-directional RNN layer its input is a composition of the output from the previous layer evaluated for the forward and backward pass. For example, to evaluate time-step (x_1) in the second layer of a deep RNN model, we need the output from layer 1 evaluated for x_1 in the forward pass and the output for x_N in the backward pass.

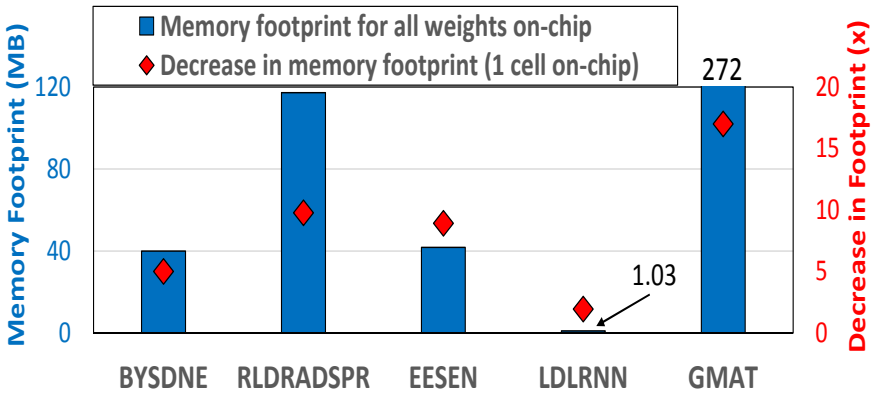


Figure 4.1: Amount of memory required to store the synaptic weights on-chip for several LSTM networks. Right y-axis shows the reduction in storage requirements obtained by keeping a single cell on-chip.

Regarding horizontal computations, they are done by first evaluating a whole layer for all the time-steps of the input sequence before proceeding with the evaluation of the next layer. Note that if the weights of a given layer are kept locally in on-chip memory, they can be reused to evaluate each time-step of the input sequence without the need to access the main memory multiple times. Hence, this approach is effective energy-wise, and we employ it on E-PUR. A drawback of this approach is that several intermediate results are generated since the output of a given layer for each time-step must be stored, as it will be used to compute the next RNN layer. We describe in section 4.2 the solution employed to mitigate this problem.

As described above, storing the entire RNN model on-chip is unfeasible for many RNN applications due to their large memory requirements to achieve high accuracy. For instance, the GMAT [110] RNN model for machine translation requires more than 256 megabytes of memory, as we can see in Figure 4.1. To alleviate this problem, we propose a cost-effective trade-off between main memory accesses and on-chip memory storage. It is based on the observation that the input sequences of RNN networks tend to contain a large number of elements and that to evaluate a given RNN layer horizontally, only the weights for that particular layer are used when computing the whole input sequence. We exploit this characteristic of RNNs to design the memory system of E-PUR, providing on-chip memory capacity to store only the weights of a single RNN layer. Note that, as seen in Figure 4.1, the storage requirements are reduced by 7x on average, although this comes at the expense of higher off-chip memory traffic. Nonetheless, this trade-off is necessary to support larger and deeper models since keeping them on-chip is unfeasible due to their large memory footprint. Also, to further reduce memory footprint weights and input sequences are converted to a lower precision (i.e., 8-bits) using linear quantization.

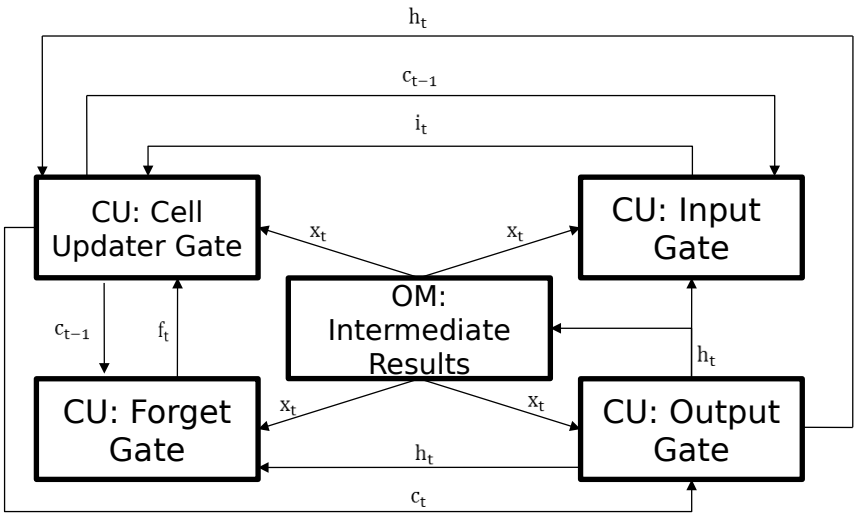


Figure 4.2: Overview of E-PUR architecture, which consists of four computation units (CU) and an on-chip memory (OM) for intermediate results. Each CU is mapped to one of the gates in an LSTM cell. However, a similar mapping is done when evaluating GRU cells.

4.2 E-PUR Processing Unit

4.2.1 Overview

Figure 4.2 shows the main components of E-PUR. It is composed of four computation units (CUs), which have several communication links among them. Each of these four hardware units is tailored to the computation of one of the four LSTM gates (i.e., forget gate, input gate, cell updater gate, and output gate), as shown in Figure 4.2. The reason for this one-to-one gate-to-CU mapping is that exchanging information between LSTM gates is not needed for most of the cell state computation. Regarding the mapping of the gates on a GRU cell, it is done in the following manner. The update gate is mapped to the CU tailored to the LSTM's input gate. On the other hand, the reset gate is assigned to the CU for the LSTM's forget gate. Finally, the GRU cell's new hidden state is computed on the CU, which is tailored to the LSTM's update gate. Note that the CU employed to evaluate the LSTM's output gate is not used when evaluating GRU cells, but it can be power gated so it does not drain any power. E-PUR is designed to deliver the highest efficiency for LSTM networks, as they represent by far the most popular RNN architecture, but it is flexible enough to support GRU networks.

The computations on a gate of an RNN cell are mainly dominated by the calculation of the matrix-vector multiplications detailed in Sections 2.2.3 and 2.2.4. Note that each gate performs exactly two matrix-vector multiplications (i.e., two dot products for each neuron) per time-step of the input sequence and, therefore, the total computation is well balanced among the gates. However, a minimal amount of information is shared among CUs at the end of the cell state calculation to gather the necessary data for its update. As shown in Figure 4.2, for an LSTM cell both input and forget gates send their result to the cell updater gate, whereas the output gate consumes the result

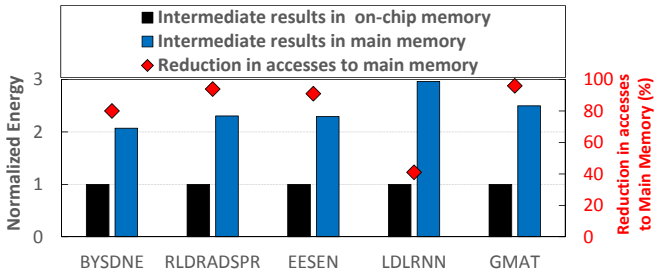


Figure 4.3: Total energy consumption by storing intermediate results in on-chip memory versus main memory. Right y-axis shows the reduction in accesses to main memory.

produced in the cell updater gate. Moreover, after the cell updater gate updates the cell state, the new cell state value is sent to the input and forget gates. Note that the hidden state of a GRU cell is computed in a similar manner.

For the rest of this chapter, we describe the operations performed by E-PUR on an RNN cell employing an LSTM cell as an example. However, note that the same principles apply to GRU cells since their computations are very similar to the LSTM's calculations.

In E-PUR, an RNN model is evaluated horizontally (i.e., one layer after the other) for a given input sequence. Thus, the intermediate results produced by one layer for an entire input sequence must be saved in memory because of the data dependencies. There are two main alternatives to store this information: a dedicated on-chip memory (OM) or main memory. In Figure 4.3, we show the normalized energy consumption and the reduction in accesses to the main memory for some LSTM applications using both approaches. As we can observe, using a dedicated on-chip memory consumes, on average, 2.4x less energy than storing/loading the intermediate results continuously to/from main memory since, on average, 77% of the accesses to main memory are avoided. Therefore, this is the adopted solution in E-PUR. This dedicated on-chip memory is divided into two parts of equal size. One part is used to store the output results produced in the current layer, and the other one is used to keep the results produced in the previous layer.

4.2.2 Computation Unit

The Computation Unit is the hardware structure that implements the formal model of an RNN cell, described in Equations 2.4 and 2.5. It is composed of two main components: the Dot Product Unit (DPU) and the Multifunctional Unit (MU). The DPU, shown at the top of Figure 4.4, performs the necessary dot product operations in a gate, which is the most time-consuming part. Note that our design employs dot products over matrix-matrix multiplications to simplify the hardware. Also, since the weights and input sequences are quantized, dot products are computed using integer operations. The MU, shown at the bottom of Figure 4.4, performs the rest of operations, such as activation functions and linear quantization. These operations are evaluated using floating point arithmetic. In addition to these components, two memory buffers are used to store the input sequence and the synaptic weights for each gate in the RNN cell. Note that the same weights are reused for each recurrent execution of an RNN cell.

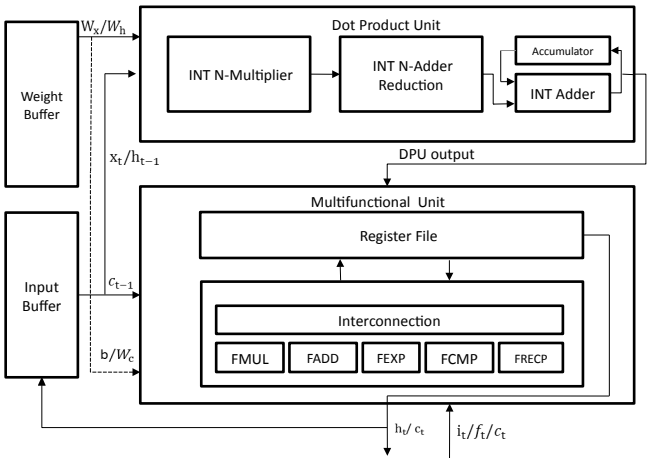


Figure 4.4: Structure of a Computation Unit.

The Dot Product Unit

The DPU performs an integer (INT) dot product between two vectors of length M by splitting them into K sub-vectors of size N. On each cycle, this unit executes the following steps. First, two size N sub-vectors are loaded from two different on-chip scratchpad memories: the Weight Buffer and the Input Buffer. The former keeps all the synaptic weights (for the recurrent and forward connections) of a given layer. The latter stores either the input time-step x_t or the previous output h_{t-1} of the layer being evaluated. Next, the N-elements INT Multiplier performs an element-wise multiplication of the two sub-vectors. Then, the resulting vector is sent to the INT N-elements Reduction Adder, to sum together all its elements, which takes $\log_2(N)$ cycles. Finally, the resulting value is added to the value stored in a register called Accumulator (24 bits), which accumulates the partial dot product until the results of all K sub-vectors are added together.

As shown in Equations 2.4 and 2.5, to evaluate a neuron in a given gate, two dot product operations are required: one takes x_t as input vector and the other one takes h_{t-1}. Then, the resulting output values of these two operations are added. In the Computation Unit, these two dot product operations are computed sequentially for each neuron, so that the latter is automatically added to the result of the former in the Accumulator register. Then, the resulting value is sent to the Multifunctional Unit (MU), which performs the remaining operations depending on the gate. Note that when a value is sent to the MU, the DPU does not wait until the MU finishes. Instead, it proceeds with the evaluation of the remaining neurons since they do not depend on the previous ones.

The Multifunctional Unit

The Multifunctional Unit (MU) is a hardware component whose activity depends on the Computation Unit (i.e. input gate) where it is located. One input to the MU is the DPU output value, which corresponds to neuron's evaluation for forward and recurrent connections. On the other hand, some of the operations performed in a particular MU may require values produced in other

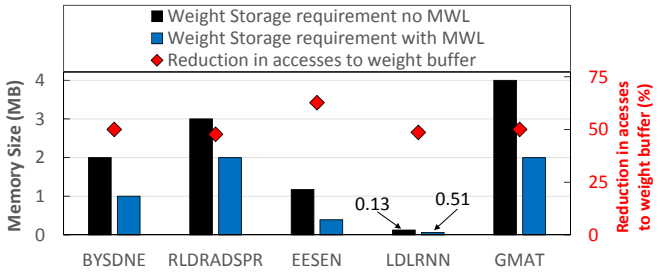


Figure 4.5: Synaptic weights memory requirements for a single LSTM cell with and without MWL. Right y-axis shows the reduction in accesses to the weight buffer.

Computation Units, as explained in Section 4.2.1.

As shown in Figure 4.4, an MU is composed of a register file, an interconnection network and several floating point and integer units that implement basic operations: multiplication, addition, division, comparison and exponential. Moreover, the previous cell state (i.e. c_{t-1} for the previous time-step in the input sequence) comes through the Input Buffer.

In Table 4.1 we detail the basic steps performed by the four MUs once the output data from the DPUs is available. For the sake of simplicity, we assume a single cycle per operation and data transfer in Table 4.1. Note that for the evaluation we use Synopsys Design Compiler to set realistic latencies for the different operations and data transfers, as reported in Table 3.1. MUs are not in the critical path, since the DPU operations are more time consuming and, thus, there is slack to accommodate multi-cycle latencies for MU operations. Moreover, note that the computation of the sigmoid and hyperbolic tangent is split into several operations and it takes multiple cycles.

The MUs for the input and forget gates perform very similar operations: they apply the sigmoid function to the result received from the DPU. After this, the resulting value is sent to the MU of the cell updater gate, which uses this information to proceed with the computation of the cell state, i.e. c_t, and, then, it applies the hyperbolic tangent function to this value. Once this information is computed, it is sent to the MU of the output gate, which computes and quantizes the k^{th} element of the output vector, i.e. h_t, corresponding to the current time-step of the input sequence (i.e. x_t).

In an MU, for a given element of the output vector (i.e. h_k) linear quantization is applied using Equations 2.6. We employ a precision of 8-bits since it does affect the model accuracy. We empirically determine α for the first layer of each network and for other layers α has a value of two since the input values and weights are between 1 and -1. In addition, the values received from the DPU are converted back to floating point, multiplying by $1/\beta$, before computing the activation function (i.e. sigmoid). Note that the weights are quantized offline and to perform quantization we include the required functional units (i.e, rounding, integer casting) on each MU.

Finally, once h_k is computed and quantized, the value h'_k is sent to the Input Buffer of all the Computation Units. In addition, it is sent to the dedicated on-chip memory where it is stored to be consumed by the next layer, as described in Section 4.2.1. Communication between MUs is performed by dedicated links, as shown in Figure 4.2.

Table 4.1: Steps of the Multifunctional Units for a single data element. DPU_O is the output of a DPU for a given CU.

stage	0	1	2	3	4	5	6	7
Input Gate	$R_0 = DPU_O$	$R_0 = -R_0$	$R_0 = e^{R_0}$	Sigmoid function $R_0 \mathrel{+}= 1$	$R_0 = \frac{1}{R_0}$	send i_t		
Forget Gate	$R_0 = DPU_O$	$R_0 = -R_0$	$R_0 = e^{R_0}$	Sigmoid function $R_0 \mathrel{+}= 1$	$R_0 = \frac{1}{R_0}$	send f_t		
Cell Updater Gate	$R_0 = DPU_O$	Hyperbolic tangent function $R_1 = -R_0$ $R_0 = e^{R_0}$	$R_1 = e^{R_1}$	$R_1 = R_0 - R_1$ $R_0 = R_0 + R_1$	$R_0 = \frac{R_1}{R_0}$	recv $i_t \& f_t$	$R_0 = R_0 * i_t$ $R_1 = f_t * c_{t-1}$	$R_0 = R_0 + R_1$
Output Gate	$R_0 = DPU_O$	wait c_t	wait c_t	wait c_t	wait c_t	wait c_t	wait c_t	recv c_t

stage	8	9	10	11	12	13	14	15
Cell Updater Gate	$R_1 = -R_0$	Hyperbolic tangent function $R_1 = e^{R_1}$ $R_0 = e^{R_0}$	$R_1 = R_0 - R_1$	$R_0 = \frac{R_1}{R_0}$ $R_0 = R_0 + R_1$	send $\phi(c_t)$			
Output Gate	$R_0 = R_0 + R_1$	$R_0 = -R_0$	$R_0 = e^{R_0}$	$R_0 \mathrel{+}= 1$	Hyperbolic tangent function $R_0 = \frac{1}{R_0}$	$h_t = R_0 * R_1$ recv $\phi(c_t)$	$R_1 = \phi(c_t)$	

4.3 MWL: Maximizing Weight Locality

As shown in Figure 4.5, on-chip memory requirements to store the synaptic weights are still quite significant for some applications (i.e. GMAT), despite the optimizations proposed in Section 4.1. In order to further improve energy consumption and reduce on-chip memory requirements, we propose a technique that maximizes temporal locality of the accesses to the weights, which are performed for each layer. We call this technique Maximizing Weight Locality (MWL). The key observation is that forward connections (i.e. their inputs come from the previous layer) can be processed in any order since all the output results from the previous layer are available. Therefore, E-PUR processes forward connections in an order that improves temporal locality. The idea is that in a given gate, instead of completely evaluating all the neurons for a single element (x_t) of the input sequence, the evaluation for all the neurons is split in two steps. In the fist step, all the neurons are evaluated using as input the forward connections for the whole input sequence (i.e, x_t, .., x_n) and the intermediate results are saved. For the second step, MWL proceeds with the computation of all neurons for the recurrent connections (i.e, h_{t-1}, .., h_{n-1}). Note that in this case, the evaluation must be done in sequence since data dependencies in the recurrent connections impose strict sequential order.

With this approach, E-PUR reuses a small subset of the weights, those corresponding to a particular neuron, at extremely short distances. Note that for a given neuron, once it is partially computed for all elements of the input sequence, its corresponding weights will no longer be required and, thus, they can be evicted from on-chip memory. Therefore, while processing forward

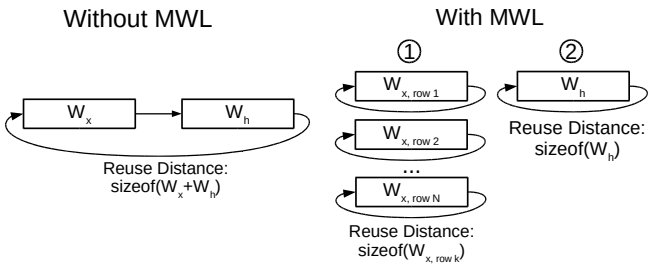


Figure 4.6: Reuse distance for the accesses to the weight information.

connections, E-PUR only requires on-chip storage for the forward weights of a single neuron at a time, significantly reducing on-chip storage requirements and energy consumption. As shown in Figure 4.5, the storage requirements for the weights are reduced by approximately 50% on average. Note that recurrent connections are evaluated as usual and, hence, all the associated weights for a given layer must be stored on-chip to avoid excessive accesses to off-chip memory.

The drawback of MWL is that it requires additional memory to store the partial evaluations of all neurons on a given layer. In the design of E-PUR, presented in Section 4.2, neurons in a cell are completely evaluated for an element in the input sequence before proceeding to the next input element. Therefore, only the final output vector of a cell, h_t, has to be stored in a memory buffer. On the other hand, with MWL, the neurons are first partially evaluated for all the elements in the input sequence, by operating exclusively on the forward connections. In this case, the partial evaluations for the neurons in each of the four gates must be stored, since later they have to be merged with the result of evaluating the recurrent connections, in order to produce the final output. This requires an increase in on-chip storage requirements for intermediate results, but this overhead is minimized applying linear quantization to the partial output results. Next subsections provide further details on the implementation and trade-offs of MWL.

Prioritize Forward Connections

The conventional way to evaluate the input sequence in a layer is by performing all the necessary computations of the current element in the input sequence before starting with the next one. It implies the evaluation of both forward and recurrent connections in each layer. However, by following this order, the temporal locality to access the weights from each gate is suboptimal. As we can see in the left part of Figure 4.6, the reuse distance of a weight access is equal to adding the size of the two weight matrices, i.e. W_x and W_h. This has a direct impact on storage requirements, since a longer reuse distance requires a larger on-chip memory to hold the weights in order to avoid expensive off-chip memory accesses.

MWL improves temporal locality in the weight accesses by changing the evaluation order of the two feed-forward networks across the entire input sequence in a given layer. It is based on the observation that all feed-forward networks that take x_t as input vector, i.e. those that contain forward connections, do not depend on the previous output of the layer, as we can see in Equations 2.4. Therefore, we improve temporal locality by partially evaluating all the neurons in a layer for the entire input sequence and then proceeding with the recurrent connections (h_{t-1}),

instead of sequentially evaluating the neurons in the layer for x_t and h_{t-1} and then proceeding with x_{t+1} and h_t. This reduces the storage requirements to the size of a single feed-forward network, as seen in Figure 4.5.

Note that for a given neuron in a cell, its computations use the same subset of weights (i.e, a single row from the weight matrix of the feed-forward network), therefore the reuse distance is reduced to a single row of the feed-forward matrix, as we can see in the middle part of Figure 4.6. Henceforth, we store them in a small buffer (i.e. 4KB), thus, avoiding to access the weight buffer for the forward connections. As a result, as shown in Figure 4.5, the accesses to the weight buffer are reduced by 50% on average.

Finally, after the partial evaluation of the forward connections for all the neurons in a layer, the evaluation for recurrent connections is performed as explained in Section 4.2, i.e. the next input is not evaluated until the results of the current input are computed, to respect data dependencies (right part of Figure 4.6).

Storage of the Intermediate Results

The dedicated on-chip memory for intermediate results (see Section 4.2.1) is dimensioned to hold the final outputs (i.e. h_t) for a given layer, which are produced by the output gates in each cell. When using MWL, the temporal values produced by each gate while evaluating forward connections must be saved for the entire input sequence since the MUs will need these values to compute the final outputs, as explained above. Therefore, the main drawback of this technique is the extra storage requirements for these intermediate values, which is equal to four times the memory needed to store the h_t outputs, because intermediate values are produced in the four gates.In order to deal with this issue, we apply linear quantization using 8 bits to each of the temporal value generated on each gate. Note that the inputs (x_t) and the forward connections (W_h) are quantized using 8 bits. Also, for a given neuron n_k with partial output (i.e. o_k), o_k is first computed using integer arithmetic. Then, the final value of o_k is a 24-bits number which it is quantized using 8-bits.

Finally, for a given neuron k with partial output (i.e. o_k) produced in MWL, its quantized value (i.e. o'_k) is stored in the on-chip memory for intermediate results. After all the partial outputs (o_k) for all the neurons are computed, recurrent connections are evaluated as explained in section 4.3. However, before computing the final output for a given gate in a cell, the previous quantized values must be converted back to floating point numbers and added to the result of evaluating the recurrent connections.

4.4 Experimental results

In this section, we present the evaluation of E-PUR, our processing unit for RNNs. We follow the methodology presented in 3. Moreover, we employed some of the networks shown in Table 3.4 as benchmarks. Also, the hardware parameter of E-PUR and E-PUR with MWL are shown in Figure 4.2.

Table 4.2: Hardware parameters for E-PUR and E-PUR with MWL.

Parameter	E-PUR	E-PUR+MWL
Technology	28 nm	28 nm
Frequency	500 MHz	500 MHz
Intermediate Memory	1.5 MB	2.5 MB
Weights Memory	2 MB per CU	1 MB per CU
Inputs Memory	8 KB per CU	4 KB per CU
DPU Width	16 operations	16 operations
MU Operations	cycles: 2 (ADD), 4 (MUL), 5 (EXP)	
MU Communication	2 cycles	2 cycles
Peak Bandwidth	30 GB/s	30 GB/s

For comparisons, we also implemented MWL in software for the Tegra X1 (Tegra X1+MWL) to analyze the benefits of a software-only implementation. We used CUDA to implement this version and employed kernel fusion [105] to merge the processing of different gates in one kernel, avoiding excessive number of API calls, which represent a significant overhead in this platform.

The baseline configuration used for comparison purposes is a cuDNN implementation running on a mobile NVIDIA Tegra X1 platform. The configuration labeled as E-PUR throughout this section consists of our first design presented in Section 4.2, whereas the configuration E-PUR+MWL includes our technique for improving the temporal locality of the weights described in Section 4.3. First, we present the energy reduction achieved by these two configurations with respect to the Tegra X1. Second, the performance improvement over the baseline is analyzed. Third, the power consumption for each of these configurations is shown. Fourth, we present the total area required by E-PUR. Finally, we analyze the performance of a software-only implementation of MWL.

Figure 4.7 shows the energy reduction. On average, E-PUR and E-PUR+MWL achieve 58x and 88x energy reduction respectively. All the LSTM networks show large improvements of at least 21x reduction in energy consumption. A remarkable case is LDLRNN, for which E-PUR reduces the total energy by 156x and 222x, respectively. The reason for this large energy reduction is that LDLRNN has fewer outputs per layer, i.e. smaller number of neurons, which means that the matrix-vector multiplications require less number of operations and, also, less memory accesses are done to fetch the weights or intermediate results. This penalizes Tegra X1 platform because the ratio between computations in the GPU and other related tasks (e.g., GPU synchronization, CPU work, etc.) is smaller. Note that for E-PUR most of the energy savings come from avoiding accesses to main memory to load/store intermediate results and weights. In the case of E-PUR+MWL, energy savings come from avoiding accesses to the on-chip memory for weights by 50% on average.

Figure 4.8 shows the energy breakdown for the two configurations of E-PUR. The different components of E-PUR are grouped into "scratchpad memories", which includes all the on-chip memories, and "operations", which includes the pipeline components, such as the functional units. Since on-chip memory requirements and number of memory accesses are significant, the overall energy consumption is dominated by the dynamic and static energy of on-chip memories, which accounts by around 95% on average. Because MWL reduces the dynamic accesses for the weight buffer by 50% on average, the dynamic energy due to on-chip memories is reduced in 25% on average

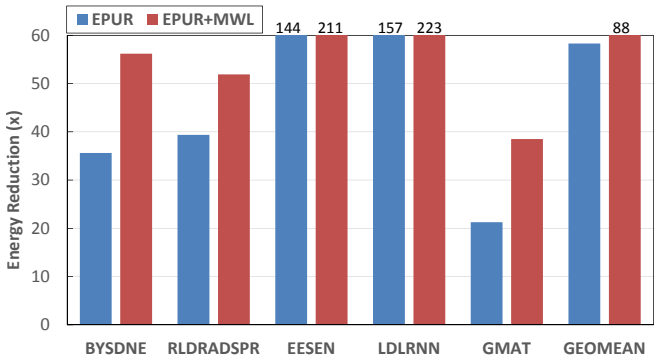


Figure 4.7: Energy reduction of E-PUR with respect to the Tegra X1.

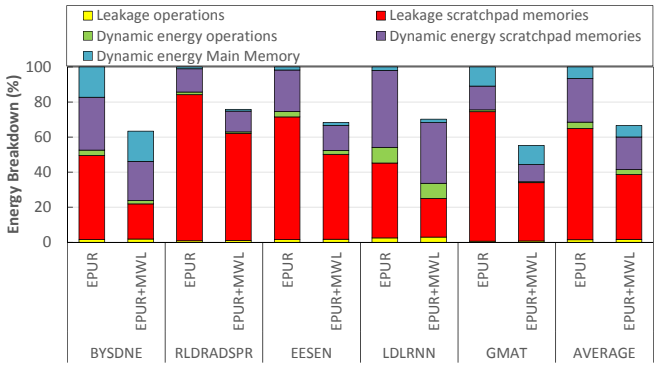


Figure 4.8: Energy breakdown for E-PUR and E-PUR+MWL.

for E-PUR+MWL. Note that the energy consumption due to scratchpad memories is not reduced by 50% since there is an increase in memory accesses to the on-chip memory for intermediate results. In the case of the leakage due to on-chip memories, after applying MWL, it is reduced by 44% on average. This saving comes from the reduction in storage requirements to store the weights for the forward connections. Regarding the energy consumption due to the operations, it ranges between 1% and 3% of the total energy for both configurations. Henceforth, after applying MWL the energy consumption is reduced by 33.4% on average.

Figure 4.9 shows the speedups for different LSTM networks. On average, the speedup achieved by E-PUR over Tegra X1 is 6.8x. E-PUR performance improvements come from hiding memory latency (i.e, loading/storing is overlapped with computations), reducing off-chip memory accesses, and featuring a custom pipeline tailored to LSTM computation.

Regarding E-PUR+MWL, there is no performance improvement against the baseline since the order in which MWL evaluates the neurons does not change the final execution time. Note that in MWL the number of operations to evaluate a given neuron is equal to the number of operations for the conventional order. However, because the evaluation of the recurrent connections for a given neuron is postponed until all forward connections are evaluated, the latency to evaluate a single neuron increases but the latency to produce the final output sequence does not change. Finally, for speech recognition applications, E-PUR achieves real-time performance by a large margin, running 30x and 5x faster than real-time for EESEN and RLDRADSPR respectively.

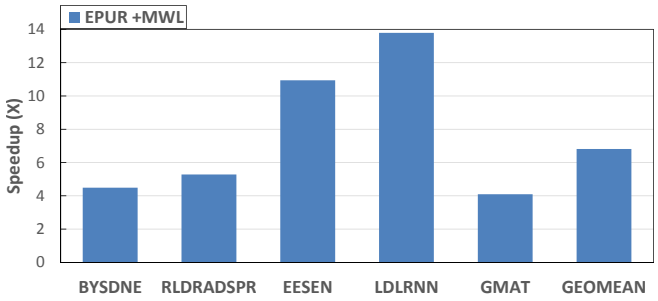


Figure 4.9: Speedups achieved by E-PUR over Tegra X1.

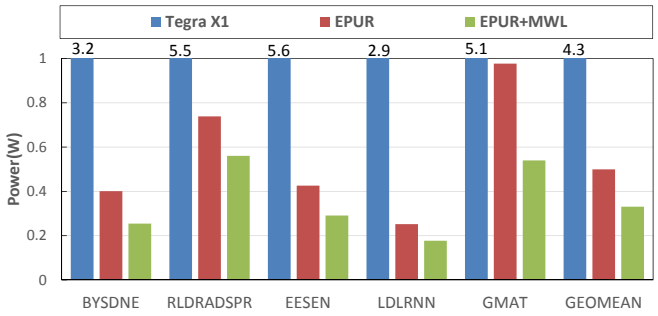


Figure 4.10: Power dissipation for E-PUR, E-PUR+MWL, and Tegra X1.

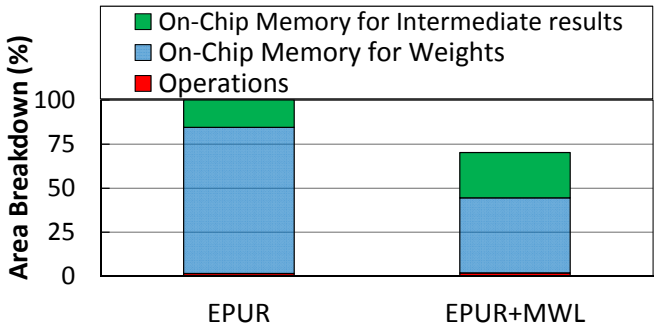


Figure 4.11: Normalized Area breakdown for E-PUR and E-PUR+MWL.

On the other hand, power dissipation is shown in Figure 4.10, which includes the total power for Tegra X1 and the two configurations of E-PUR. As it can be seen, E-PUR+MWL dissipates about 10x lower power than Tegra X1 on average.

Regarding area, E-PUR requires a total area of 31.3 mm^2, whereas the total area of E-PUR+MWL is 22 mm^2. As depicted in Figure 4.11, the component with larger contribution to the total area is the on-chip memory for the synaptic weights, which is reduced by 48.6% when MWL is applied. Note that with MWL the memory requirements to store intermediate results increase. Hence, the overall saving in area due to MWL is 30%.

Finally, Figure 4.12 shows the speedup and energy reduction of the Tegra X1+MWL, i.e. MWL implemented in software, with respect to the baseline. On average, it provides 7.8% speedup,

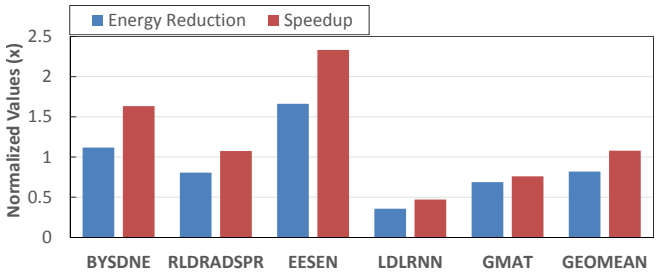


Figure 4.12: Speedup and normalized energy achieved by Tegra X1+MWL over Tegra X1.

increasing energy consumption by 25%. Overall, Tegra X1+MWL achieves better performance than the baseline but since the on-chip memories of Tegra X1 are fairly smaller than the ones included in E-PUR, the effectiveness of Tegra X1+MWL for RNNs is constrained due to the increase in off-chip memory traffic. As a result, the baseline implementation consumes less energy than Tegra+MWL.

4.5 Related Work

Proposals for LSTM networks acceleration have been presented in [43, 41, 62]. Although these accelerators achieve higher performance per watt than CPUs and GPUs, they are not designed for low-power mobile devices since their power dissipation ranges from 19 W to 41 W. However, E-PUR dissipates a peak power of 330 mW, which is amenable for low-power mobile devices.

Chang et al. [17] present a low-power accelerator targeting the mobile segment. It implements a small LSTM network (2 layers, 128 neurons) and dissipates 1.9 W. In this work arithmetic operations are done using fixed-point Q8.8 data format, thus an accuracy loss of 7.1% is aggregated. On the contrary, E-PUR uses 8-bit linear quantization to store the weights, matrix multiplications are perform using integer operations, and it does not aggregate accuracy loss. In addition, E-PUR supports larger network models for a wide variety of application domains. Note that scaling up the aforementioned accelerator presented in [17] to support larger LSTM networks would require a significant increase in local storage capacity or in main memory traffic, and both alternatives would come at a high overhead in energy consumption.

Another low-power LSTM accelerator is presented in [61], this system consumes 9 W and supports larger models by using aggressive weight quantization. External DRAM traffic is completely avoided by storing the quantized weights in a local on-chip memory of 2 Mbytes. However, this quantization comes at the expense of non-negligible accuracy loss. For speech recognition, Word Error Rate increases from 13.5%, using 32-bit floating point, to 15.1% and 20.2% when using 6-bit and 4-bit quantization respectively. Furthermore, larger and more accurate models cannot be stored in its local memory even with the 4-bit quantization. For example, EESEN requires more than 5 Mbytes when using 4 bits per weight. E-PUR uses 8-bit quantization to reduce memory footprint with not impact on accuracy.

TPU is another state-of-the-art accelerator for neural networks [51]. It includes a systolic array of processing elements (PEs) and on-chip memory for weights and activations. TPU is mainly

tailored to CNNs, and thus it has low hardware resource utilization (i.e., 18%) when evaluating RNNs. On the contrary, E-PUR has nearly 100% hardware resource utilization. Furthermore, TPU's power dissipation is 200 watts [37], whereas E-PUR has a peak power dissipation of 330mW. Recently, Edge TPU [38], a low power version of TPU, has been proposed. It is tailored to CNNs and fully-connected networks, whereas this work focuses on RNNs. Edge TPU dissipates 0.5 watts for each TOPS.

BrainWave [29] is a highly optimized accelerator targeting DNNs. In this accelerator, matrix-matrix multiplications are performed using a vast array of matrix-vector multipliers. Similar to E-PUR, BrainWave includes multi-functional units to compute activation functions. However, as TPU, BrainWave's resource utilization is only 3.5% when evaluating RNNs. On the contrary, E-PUR achieves nearly 100% utilization. Finally, its peak power computation is 125W, which is not amenable to low power devices.

Regarding pruned RNNs, MASR [42] is an RNN accelerator tailored to sparse models. It employs pruning to improve energy-efficiency. Also, it does not store or compute null values. ESE [43] and EIE [44] are another RNN accelerators for sparse models. ESE targets speech recognition applications, whereas EIE targets image captioning. Also, EIE is tailored to CNNs. Both of these proposals employed aggressive pruning to reduce the model size. On the contrary, E-PUR focuses on dense models and maximizes weight locality to improve energy efficiency. Moreover, E-PUR is designed to support applications from several domains.

Regarding the work in [19], E-PUR without MWL is similar to a weight stationary architecture applied to LSTMs since it loads all weights for given layer in on-chip memory, holding them until all associated computations are performed. However, MWL is different since it aims at further reducing the reuse distances. Unlike traditional weight stationary architectures, MWL splits synaptic weights in two types: forward and recurrent. Based on the observation that forward connections can be processed in any order, whereas recurrent connections impose sequential processing due to data dependencies. Therefore, MWL evaluates forward connections in the order that maximizes temporal locality, requiring extra small on-chip storage for this stage, whereas it processes all recurrent connections on a second stage as shown in Figure 4.6. MWL greatly reduces the energy consumption of the baseline accelerator.

Finally, cuDNN [11] has been recently extended to efficiently support RNN training. E-PUR design is significantly different in multiple ways. First, cuDNN focuses on RNN training with large batch sizes, whereas E-PUR focuses on RNN inference with batch size of one, i.e. one input sequence at a time. We measured cuDNN performance for RNN inference with batch size of one and found that E-PUR achieves 6.8x speedup. cuDNN effectiveness is reduced due to the small batch size commonly used for RNN inference. Furthermore, cuDNN's optimizations to execute multiple layers in parallel cannot be applied to bidirectional LSTMs due to data dependencies.

4.6 Conclusions

In this chapter, we present E-PUR, a processing unit for RNNs that supports large RNN networks while dissipating low-power, motivated by the increasingly important role of RNN networks

in applications such as speech recognition, machine translation and video classification.

Unlike previous proposals that attempt to accommodate the entire RNN on-chip, E-PUR only provides storage for one RNN layer, whose weights are fetched once from main memory and reused for multiple recurrent executions. To further improve the memory efficiency of E-PUR, we introduce Maximizing Weight Locality (MWL), a novel technique that improves the temporal locality of the synaptic weights. Moreover, we implement MWL in software to analyze the benefits of a software-only implementation

The proposed design supports large LSTM and GRU networks of hundreds of Megabytes, while using small on-chip storage and low memory bandwidth. Our results show that E-PUR reduces energy consumption by 88x on average with respect to a modern mobile GPU, while providing 6.8x speedup.

5

Neuron-Level Fuzzy Memoization

In chapter 4, we presented E-PUR, an accelerator for RNN inference which provides high performance and energy efficiency. Although the energy savings and performance improvements of E-PUR are significant, we observe that the primary source of energy consumption is still the access to the local on-chip memories of E-PUR, which account for up to 80%. In this case, most of the memory accesses are done to fetch the weights when computing the output of a given neuron. In this chapter, we explore computation reuse techniques with the aim of avoiding accesses to the on-chip memories. More specifically, we leverage a memoization scheme where the output of a neuron is cached and reused in future evaluations, skipping the corresponding memory accesses and computations.

Memoization is a well-known optimization technique used to improve performance and energy consumption that has been used both in software [3] and hardware [35]. In some applications, a given function can be executed repeatedly but the inputs to those executions are not always distinct. Memoization exploits this fact to avoid these redundant computations by reusing the result of a previous evaluation. In general, the first time an input is evaluated, the result is cached in a memoization table. Subsequent evaluations probe the memoization table and reuse previously cached results if the current input matches a previous execution.

In a classical memoization scheme, a memoized value is only reused when it is known to be equal to the real output of the computation. However, for some resilient applications such as multimedia [6], graphics [12], and neural networks [116], this scheme can be extended to tolerate a small loss in accuracy with negligible impact in the quality of the results, and is normally referred to as fuzzy memoization.

In this work, we leverage fuzzy memoization to avoid neuron evaluations selectively and, hence, to avoid their corresponding memory accesses and computations. For fuzzy memoization to be effective, applications must be tolerant of small errors, and its hardware implementation must be

simple. In the next sections, we show that RNNs are resilient to minor errors in the outputs of the neurons. Also, we provide an efficient implementation of a fuzzy memoization scheme that requires simple hardware support.

First, to assess the feasibility of memoizing the previous output of a given neuron, we analyze the output of millions of neurons in several RNN models to determine how much the output of a neuron changes between the evaluation of consecutive time-steps. Note that for each application run, an RNN layer is executed many times to process an input sequence, which normally contains a large number of time-steps (i.e., words, images, audio frames). From this analysis, we observe that after evaluating a neuron for consecutive time-steps, its output exhibits small changes, 23% on average.

Motivated by the previous observation, we perform a second analysis to establish the potential for reusing previous computations, and thus to avoid them so that energy efficiency is improved. The results of this analysis show that the potential for reuse is between 20% and 50% with a negligible impact on the model accuracy. Therefore, we exploit this observation to build a fuzzy memoization scheme, which dynamically caches each neuron's output and reuses it whenever it is predicted that the current output will be similar to a previously computed result, avoiding the computations and the memory access for some neurons.

In the rest of this chapter, we describe our analyses and present our fuzzy memoization scheme. The organization is as follows. First, we detail a study of the neurons' output in several RNN models and show that reuse is between 20% and 50%. Second, we describe our motivations for fuzzy memoization in RNNs. Then, we present our memoization scheme. Finally, we detail the experimental results and conclusions.

5.1 Analysis of Potential for Computation Reuse

As mentioned in Chapter 2, neurons in an RNN are recurrently executed for each of the time-steps in the input sequence X. Moreover, for a time-step (x_t) the output of neuron n_k is computed by adding the result of the inner product between the forward connections and x_t, to the result of the inner product between the recurrent connections and the previous output of the neuron y_{t-1}. In this regard, the relative difference (δ) between the output of n_k between consecutive time-steps x_t and x_{t-1} is defined as follows.

$$\delta = \left| \frac{y_t - y_{t-1}}{y_t} \right| \tag{5.1}$$

In this chapter, we aim to reuse computations by caching the previous output of a neuron (y_m) and reusing it when we determine that it is similar to the current one (y_t). The similarity between y_m and y_t is defined as their relative difference (δ) which is computed using Equation 5.1. In the next subsection, we experimentally show that there is a high degree of similarity between the current and previous output of some neurons and that for some evaluations, a previously computed output could be reused with a negligible impact in the model accuracy.

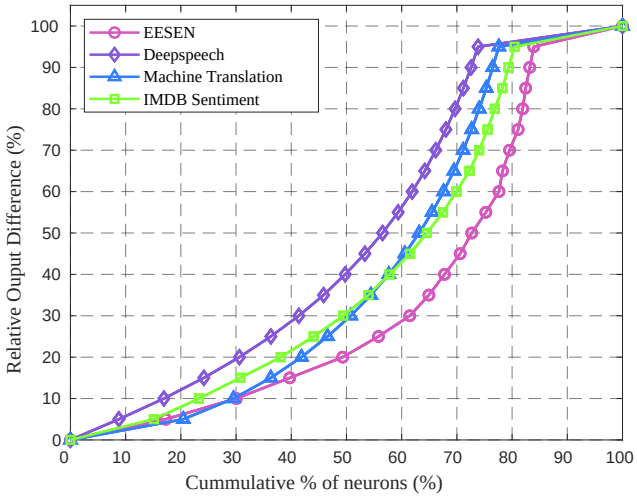


Figure 5.1: Relative change in neuron output between consecutive time-steps.

5.1.1 RNNs Redundancy and Reuse Potential

Memoization schemes rely on a high degree of redundancy in the computations. For RNNs, a key observation is that the output of a given neuron tends to change lightly between consecutive time-steps. Note that RNNs are used in sequence processing problems such as speech recognition or video processing, where RNN inputs in consecutive time-steps tend to be extremely similar. Prior work in [82] reports high similarity across consecutive frames of audio or video. Not surprisingly, our own numbers for our set of RNNs also support this claim. Figure 5.1 shows the relative difference between consecutive outputs of a neuron in our set of RNNs. As it can be seen, a neuron's output exhibits small changes (less than 10%) for 25% of consecutive input time-steps. On average, consecutive outputs change by 23%. Furthermore, RNNs can tolerate small errors in the neuron output [116]. This observation is supported by data shown in Figure 5.2, where the accuracy curve shows the accuracy loss when the output of a neuron is reused using fuzzy memoization, for different thresholds (x-axis) that control the aggressiveness of the memoization scheme.

For the study shown in Figure 5.2, we implemented the memoization scheme described in Equations 5.2, 5.3, and 5.4 . For this scheme, the relative error (δ) between a predicted neuron output (y_t^p) and a previously cached neuron output (y_m) is used as the discriminating factor to decide whether the previous output is reused, as shown in Equation 5.2. For this memoization scheme, the predicted value is provided by an Oracle predictor, which is 100% accurate (i.e., its prediction is always equal to the neuron output ($y_t^p = y_t$)). We employ this approach to evaluate the potential benefits of memoization in RNNs. As shown in Figure 5.2, neurons can tolerate a relative output error between 0.3 and 0.5 without significantly affecting the overall network accuracy (i.e., accuracy loss smaller than 1%). On the other hand, the reuse curve shows the percentage of neuron computations that could be avoided through this memoization with an Oracle predictor. Note that by allowing neurons to have an output error between 0.3 to 0.5, at least 30% of the total network computations could be avoided.

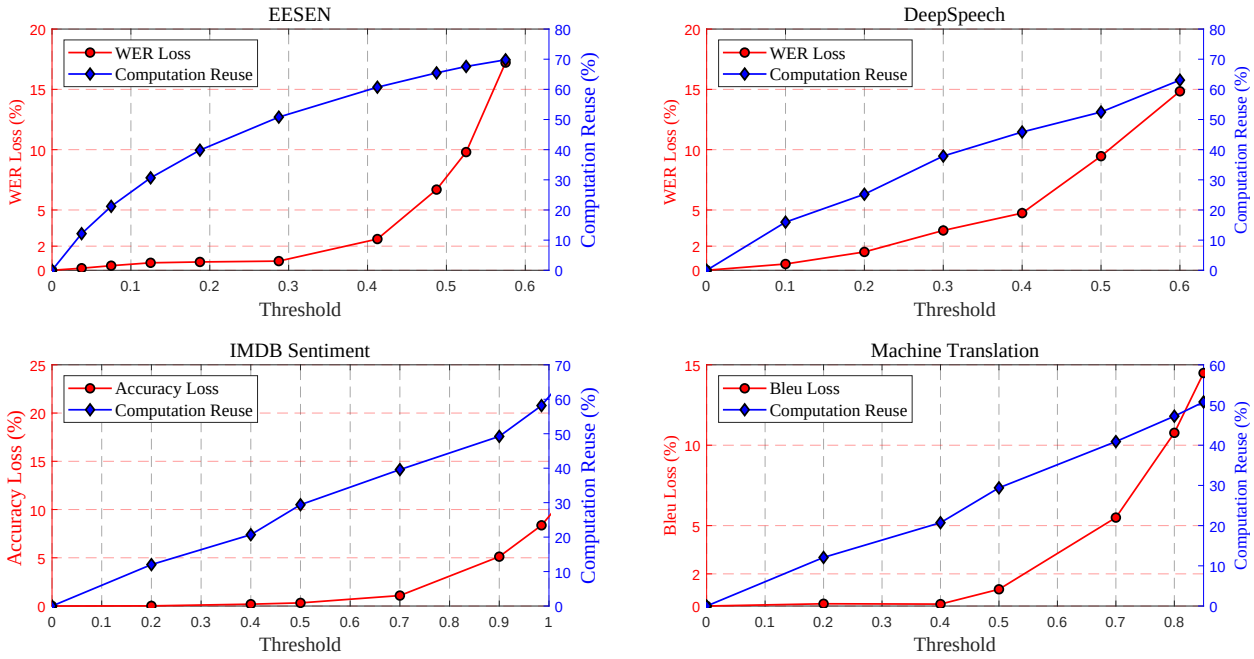


Figure 5.2: Accuracy loss of different RNNs versus the relative output error threshold using an oracle predictor. If the difference between the previous and current output predicted is smaller than the threshold, the memoized output is employed instead of calculating the new one.

$$\delta = \left| \frac{y_t^o - y_m}{y_t^o} \right| \tag{5.2}$$

$$y_t = \begin{cases} y_m & \text{if } \delta <= \theta \\ y_t^o & \text{otherwise,} \end{cases} \tag{5.3}$$

$$y_m = \begin{cases} y_t^o & \text{if } \delta > \theta \\ \text{not updated} & \text{otherwise,} \end{cases} \tag{5.4}$$

Figure 5.3: Neuron Level memoization with Oracle Predictor. y_t is the neuron output. y_m corresponds to the memoized evaluation and y_t^o is the output of the Oracle predictor. δ, θ are the relative error and the maximum allowed output error respectively.

Finally, to achieve significant savings, a memoization scheme must add a small overhead to the system. In our case, the critical challenge is approximating the behavior of the Oracle predictor with simple hardware that decides when memoization can be safely applied with a minor impact on overall RNN accuracy. We describe an effective solution in the next section.

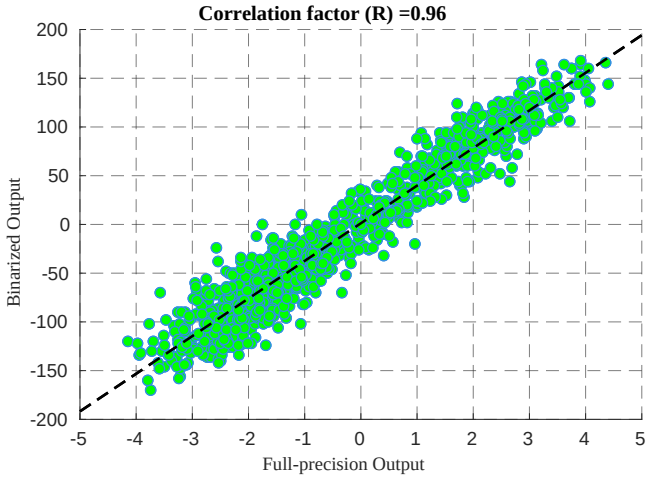


Figure 5.4: Outputs of the binarized neurons (y-axis) versus outputs of the full-precision neurons (x-axis) in EESEN, an RNN for speech recognition. BNN and RNN outputs are highly correlated, showing a correlation coefficient of 0.96.

5.1.2 Binary Network Correlation

A key challenge for an effective fuzzy memoization scheme is to identify when the next neuron output will be similar to a previously computed (and cached) output. Note that having similar inputs does not necessarily result in similar outputs, as inputs with small changes might be multiplied by large weights. Our proposed approach is based on a Bitwise Neural Network (BNN). In particular, each fully-connected neural network (NN) is extended to an equivalent BNN, as described in Section 5.2. We use BNNs for two reasons. First, the outputs of a BNN and its corresponding original NN are highly correlated [8], i.e. a small change in a BNN output indicates that the neuron's output in the original NN is likely to be similar. Second, BNNs can be implemented with extremely low hardware cost.

Regarding the correlation between BNN and RNN, Anderson et al. [8] show that the binarization approximately preserves the dot-products that a neural network performs for computations. Therefore, there should be a high correlation between the outputs of the full-precision neuron and the outputs of the corresponding binarized neuron. We have empirically validated the dot product preservation property for our set of RNNs. Figure 5.4 shows the linear correlation between RNN outputs and the corresponding BNN outputs for EESEN network. Although the range of the outputs of the full-precision (RNN) and binarized (BNN) dot products are significantly different, their values exhibit a strong linear correlation (correlation coefficient of 0.96). On the other hand, Figure 5.5 shows the histogram of the correlation coefficients for the neurons in four different RNNs. As it can be seen, correlation between binarized and full-precision neurons tend to be high for all the RNNs. More specifically, for the networks EESEN, IMDB SENTIMENT, and DEEPSPEECH, 85% of the neurons have a linear correlation factor greater than 0.8 and for the Machine Translation network most of them have a correlation factor greater than 0.5. These results indicate that if the output of a binarized neuron shows very small changes with respect to a previously computed output, it is very likely that the full-precision neuron will also show small changes and, hence,

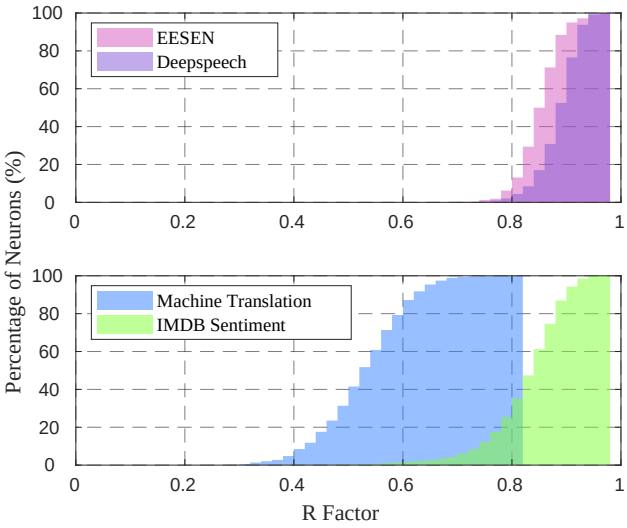


Figure 5.5: Correlation factor between the neuron output computed using full precision and the output computed with a BNN.

memoization can be safely applied.

As shown in Equation 2.9, the output of a given neuron in a BNN can be computed with an N-bit XOR operation for bit multiplication and an integer adder to sum the resulting bits. These two operations are orders of magnitude cheaper than those required by the traditional data representation (i.e., FP16). Therefore, a BNN represents a low overhead and accurate manner to infer when the output of a neuron is likely to exhibit significant changes with respect to its recently computed outputs. Further details on BNNs are provided in Section 2.3.2.

5.2 Fuzzy Memoization Scheme

The target of our memoization scheme is to reuse a recently computed neuron output, y_m, as the output for the current time-step, y_t, provided that they are very similar. Reusing the cached neuron output avoids performing all the corresponding computations and memory accesses. To determine whether y_t will be similar to y_m, we use a BNN as a predictor of reusability.

In our memoization scheme, we extend the RNN with a much simpler BNN. The BNN model is created by mirroring the full precision trained model of an LSTM or GRU gate, as illustrated in Figure 5.6. More specifically, each neuron is binarized by applying the binarization function shown in Equation 2.8 to its corresponding set of weights. Therefore, in a gate every neuron n with weights vector $\vec{w}$ is mirrored to a neuron n^b with weights vector $\vec{w}^b$ corresponding to the element-wise binarization of $\vec{w}$.

Our scheme stores recently computed outputs for the binary neuron n^b and its associated full precision neuron n. We refer to these memoized values as y_m^b and y_m respectively. On every time-step t, the binarized version of the neuron, n^b, is evaluated first obtaining y_t^b. Next, we compute the

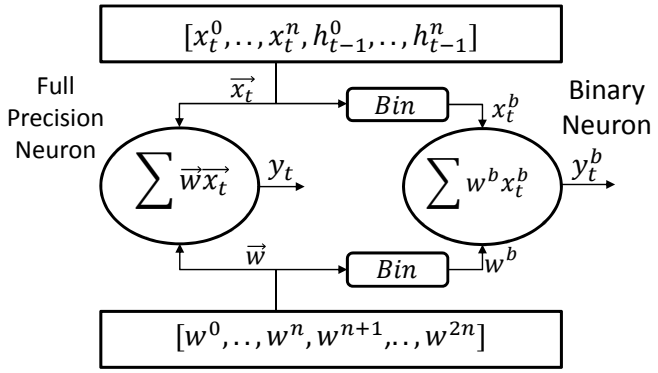


Figure 5.6: The figure illustrates how a binary neuron is created from a full precision neuron in the RNN network. *Bin* is the binarization function shown in Equation 2.8.

$$\epsilon_t^b = \left| \frac{y_t^b - y_m^b}{y_t^b} \right| \tag{5.5}$$

$$\delta_t^b = \sum_{i=m}^{i=t} \epsilon_i^b \tag{5.6}$$

$$y_t = \begin{cases} y_m & \text{if } \delta_t^b <= \theta \\ \text{evaluate neuron} & \text{otherwise,} \end{cases} \tag{5.7}$$

$$y_m = \begin{cases} y_t & \text{if } \delta_t^b > \theta \\ \text{not updated} & \text{otherwise,} \end{cases} \tag{5.8}$$

$$y_m^b = \begin{cases} y_t^b & \text{if } \delta_t^b > \theta \\ \text{not updated} & \text{otherwise,} \end{cases} \tag{5.9}$$

$$\delta_t^b = \begin{cases} 0.0 & \text{if } \delta_t^b > \theta \\ \text{not updated} & \text{otherwise,} \end{cases} \tag{5.10}$$

Figure 5.7: Neuron level fuzzy memoization with binary network as predictor. y_t, y_m correspond to the neuron current and memoized output computed by the RNN. y_t^b, y_m^b are the current and memoized output computed by the Binary Network. ϵ_t^b is the relative difference between BNN outputs. δ_t^b is the summation of relative differences in successive time-steps.

relative difference, ϵ_t^b, between y_t^b and y_m^b, i.e. the current and memoized outputs of the BNN, as shown in Equation 5.5. If ϵ_t^b is small, i.e. if the BNN outputs are similar, it means that the outputs of the full precision neuron are likely to be similar. As we discuss in Section 5.1.2, there is a high correlation between BNN and RNN outputs. In this case, we can reuse the memoized output y_m as the output of neuron n for the current time-step, avoiding all the corresponding computations. If the relative difference ϵ_t^b is significant, we compute the full precision neuron output, y_t, and update our memoization buffer, as shown in Equations 5.8, 5.9 and 5.10 so that these values can be reused in subsequent time-steps.

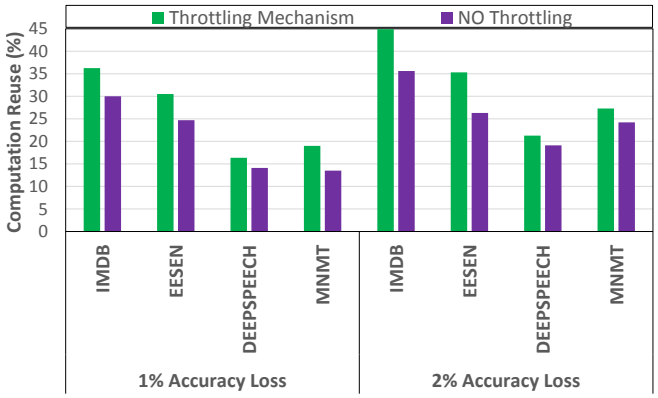


Figure 5.8: Computation reuse achieved by our BNN-based memoization scheme with and without the throttling mechanism, for accuracy losses of 1% and 2%. The throttling mechanism provides an extra 5% computation reuse on average for the same accuracy.

We have observed that applying memoization to the same neuron in a large number of successive time-steps may negatively impact accuracy, even though the relative difference ϵ_t^b in each individual time-step is small. We found that using a simple throttling mechanism can avoid this problem. More specifically, we accumulate the relative differences over successive time-steps where memoizaiton is applied, as shown in Equation 5.6. We use the summation of relative differences, δ_t^b, to decide whether the memoized value is reused. As illustrated in Equation 5.7, the memoized value is only reused when δ_t^b is smaller than a threshold θ. Otherwise, the full precision neuron is computed. This throttling mechanism avoids long sequences of time-steps where memoization is applied to the same neuron, since δ_t^b includes the differences accumulated in the entire sequence of reuses. Figure 5.8 shows that the throttling mechanism provides higher computation reuse for the same accuracy loss.

Figure 5.9 summarizes the overall memoization scheme, that is applied to the gates in an RNN cell as follows. For the first input element (x_0), i.e. the first time-step, the output values y_0^b (binarized version) and y_0 (in full precision) are computed for each neuron and stored in a memoization buffer. δ_0^b is set to zero. In the next time-step, with input x_1, the value y_1^b is computed first by the BNN. Then, the relative error (ϵ_1^b) between y_1^b and the previously cached value, y_0^b, is computed and added to δ_0^b to obtain δ_1^b. Then, δ_1^b is compared with a threshold θ. If δ_1^b is smaller than θ, the cached value y_0 is reused, i.e. y_1 is assumed to be equal to y_0, and δ_1^b is stored in the memoization buffer. On the contrary, if δ_1^b is larger than θ, the full precision neuron output y_1 is computed and its value is cached in a memoization buffer. In addition, y_1^b is also cached and δ_1^b is set to zero. This process is repeated for the remaining time-steps and for all the neurons in each gate.

Finding the threshold value

One of the key parameters in our scheme is the threshold (θ). We perform an exploration of different values of θ for each RNN model by using the training set. Then, we obtain the accuracy and degree of computation reuse for each threshold value and RNN network. We then select the

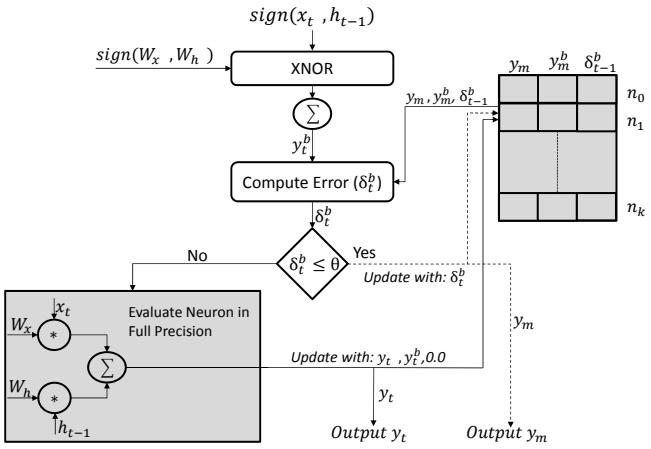


Figure 5.9: Fuzzy memoization scheme. W_x and W_h are the weights for the forward (x_t) and recurrent connections (h_{t-1}) respectively. y_t, y_m correspond to the current and cached neuron output computed in full precision. y_t^b, y_m^b are the current and cached output computed by the Binary Network. δ_t^b is the summation of relative differences in successive time-steps.

threshold value that achieves the highest computation reuse for the target accuracy loss (i.e., less than 1%) in each RNN model. Note that this process is done just once for each RNN model. Moreover, once θ is determined, it can be used for inference on the test dataset.

5.3 Hardware Implementation

We implement the proposed memoization scheme on top of E-PUR, which was detailed in Chapter 4. As described earlier, E-PUR is composed of four computational units tailored to the evaluation of each gate in an RNN cell, and a dedicated on-chip memory used to store intermediate results. In this section, we detail the necessary hardware modifications required to support our fuzzy memoization scheme.

In order to perform fuzzy memoization through a BNN, two modifications are done to each CU in E-PUR. First, the weight buffer is split into two buffers: one buffer is used to store the weight signs (sign buffer) and the other is used to store the remaining bits of the weights. Note that the sign buffer is always accessed to compute the output of the binary network (y_t^b) whereas the remaining bits are only accessed if the memoized value (y_m) is not reused. The binarized weights are stored in a small memory which has low energy cost. As a consequence of splitting the weight buffer, its area increases by less than one percent.

The second modification to the CUs is the addition of the fuzzy memoization unit (FMU) which is used to evaluate the binary network and to perform fuzzy memoization. This unit takes as input two size-T vectors (i.e., number of neurons in an RNN cell). The first vector is a weight vector loaded from the sign buffer whereas the other is created as the concatenation of the forward (x_t) and the recurrent connections (h_{t-1}).

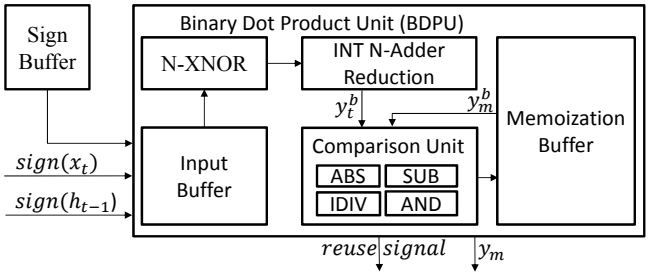


Figure 5.10: Structure of the Fuzzy Memoization Unit (FMU).

As shown in Figure 5.10, the main components of the FMU are the BDPU that computes the binary dot product and the comparison unit (CMP) which decides when to reuse a memoized value. In addition, the FMU includes a buffer (memoization buffer) which stores the δ_t^b for every neuron and the latest evaluation of the neurons by the full precision and binary networks. BNN neurons (i.e, binary dot product) are evaluated using a bitwise XNOR operation and an adder reduction tree to gather the resulting bit vector. In the CMP unit, the relative error (δ_t^b) is computed using integer and fixed-point arithmetic.

The steps followed by E-PUR to evaluate an RNN cell, described in Section 4.2, are executed in a slightly different manner to include the fuzzy memoization scheme. First, the binarized input and weight vectors for a given neuron in a gate are loaded into an FMU from the input and sign buffers respectively. Next, the BDPU computes the dot product and sends the result (y_t^b) to the comparison unit (CMP). Then, the CMP loads the previously cached values y_m^b and δ_{t-1}^b from the memoization buffer and it uses them to compute the relative error (ϵ_t^b) and the δ_t^b. Once δ_t^b is computed, it is compared with a threshold (θ) to determine whether the full precision neuron needs to be evaluated or the previously cached value is reused instead. In the case that δ_t^b is greater than θ, an evaluation in full precision is triggered. In this regard, the DPU is signaled to start the full precision evaluation which is done following the steps described in Section 4.2. After the full precision evaluation, the values y_t, y_t^b, and 0.0 are cached in the memoization table corresponding to y_m, y_m^b, and δ_t^b respectively. On the other hand, if memoization can be applied (i.e. δ_t^b is smaller than the maximum allowed error), δ_t^b is updated in the memoization table and the memoized value (y_m) is sent directly to the MU (bypassing the DPU), so the full precision evaluation of the neuron is avoided. Finally, these steps are repeated until all the neurons in a gate are evaluated for the current input element. Since LSTM or GRU gates are processed by independent CUs, the above process is executed concurrently by all gates.

5.4 Experimental Results

In this section, we present the evaluation of our memoization scheme. We employ the methodology presented in Chapter 3. Moreover, we used some of the networks shown in Table 3.4 as benchmarks, and the test set provided for each RNN. The original accuracy for each RNN is listed in Table 3.4, and the accuracy loss is later reported as the absolute loss with respect to the baseline accuracy.

Table 5.1: Configuration Parameters for E-PUR+BM.

E-PUR	
Parameter	**Value**
Technology	28 nm
Frequency	500 MHz
Intermediate Memory	6 MiB
Weight Buffer	2 MiB per CU
Input Buffer	8 KiB per CU
DPU Width	16 operations
Memoization Unit	
BDPU Width	2048 bits
Latency	5 cycles
Integer Width	2 bytes
Memoization Buffer	8 KiB

The proposed fuzzy memoization technique for RNNs is implemented on top of E-PUR and we refer to it as E-PUR+BM. First, we detail the percentage of computation reuse and the accuracy achieved. Second, we show the performance and energy improvements, followed by an analysis of our technique's area overheads.

Figure 5.11 shows the percentage of computation reuse achieved by the BNN and the Oracle predictors. The percentage of computation reuse indicates the percentage of neuron evaluations avoided due to fuzzy memoization. For accuracy losses smaller than 2%, the BNN obtains a percentage of computation reuse extremely similar to the Oracle. The networks EESEN and IMDB are highly tolerant to errors in neuron's outputs, thus, for these networks, our memoization scheme achieves reuse percentages of up to 40% while having an accuracy loss smaller than 3%. Note that, for classification problems, BNNs achieve an accuracy close to the state-of-the-art [80] and, hence, it is not surprising that the BNN predictor is highly accurate for approximating the neuron output. For DeepSpeech (speech recognition) the reuse percentage is up to 20% for accuracy losses smaller than 2%. In this network, the input sequence tends to be large (i.e, 900 elements on average). As the reuse is increased, the error introduced to the output sequence of a neuron persists for a larger number of elements. Therefore, the introduced error will have a larger impact both in the evaluation of the current layer, due to the recurrent connections, and the following layers. As a result, the overall accuracy of the network decreases faster. For MNMT (machine translation) the BNN predictor and the oracle achieve similar reuse versus accuracy trade-off for up to 23% of computation reuse. Note that, for this network, the linear correlation between the BNN and the full precision neuron output is typically lower than for the other networks in the benchmark set.

Figure 5.12 shows the energy savings and computation reuse achieved by our scheme, for different thresholds of accuracy loss. For a conservative loss of 2%, the average energy saving is 25.5%, whereas the reuse percentage is 31%. In this case, the networks DeepSpeech and MNMT show similar energy savings, whereas the networks IMDB and EESEN exhibit the largest savings since they are more tolerant to errors in the neuron output. For an extremely conservative 1% of accuracy loss, the computation reuse and energy saving are 24.2% and 18.5% on average respectively. EESEN and DeepSpeech achieve 25.32% and 12.23% energy savings respectively for a 1% accuracy

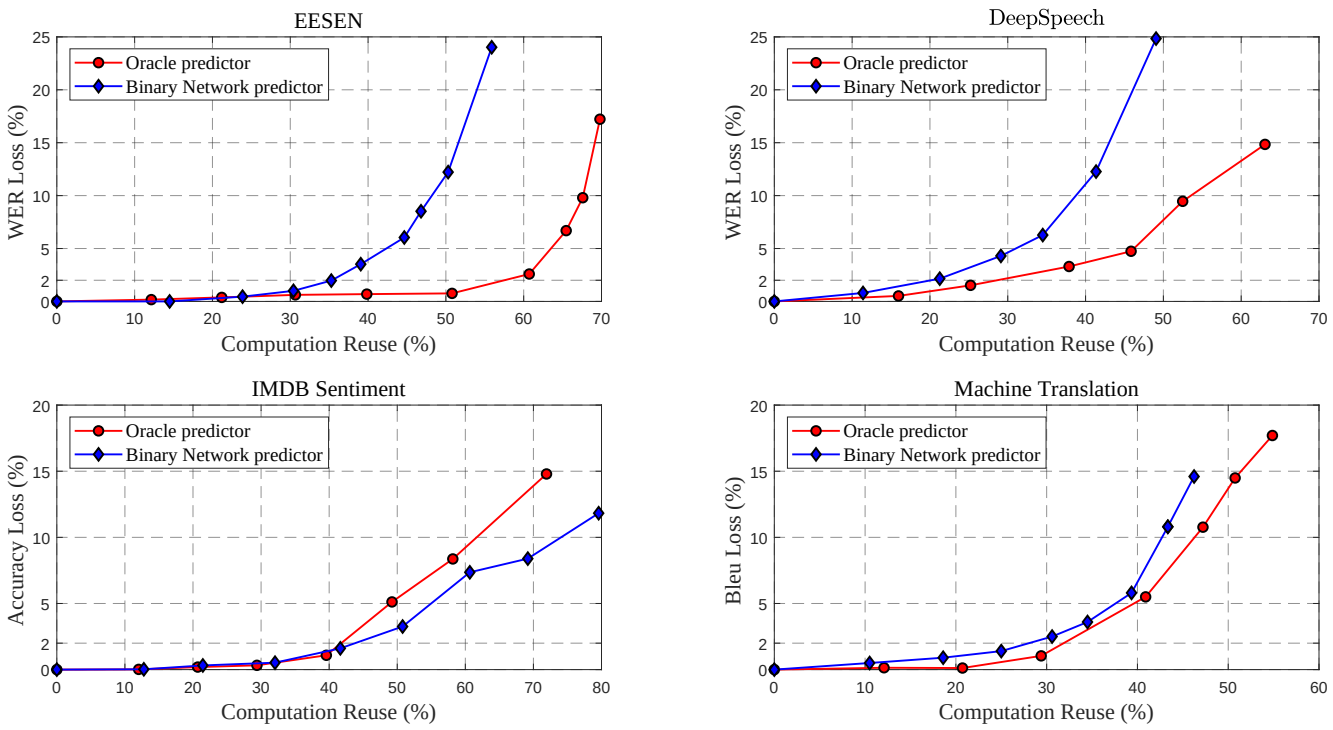


Figure 5.11: Percentage of computations that could be reused versus accuracy loss using Fuzzy Neuron Level Memoization with an Oracle and a Binary Network as predictors for several RNNs.

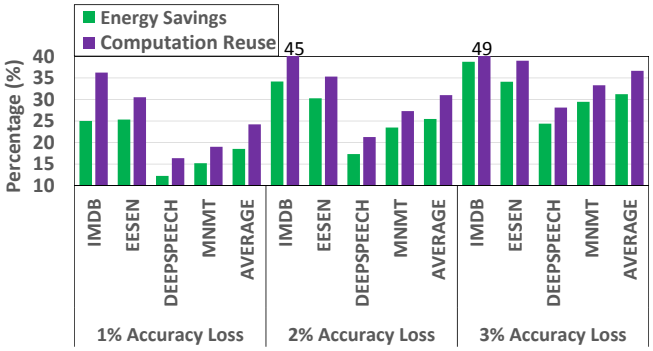


Figure 5.12: Energy savings and computation reuse of E-PUR+BM over the baseline (E-PUR).

loss. Regarding the machine translation network (MNMT), the energy savings for 1% and 2% accuracy loss are 15.17% and 23.46% respectively.

Regarding the sources of energy savings, Figure 5.13 reports the energy breakdown, including static and dynamic energy, for the baseline accelerator and E-PUR+BM, for an accuracy loss of 1%. The sources of energy consumption are grouped into on-chip memories ("scratch-pad" memories), pipeline components ("operations", i.e. multipliers), main memory (LPDDR4) and the energy consumed by our FMU component. Note that most of the energy consumption is due to the scratch-pad memories and the pipeline components and, as it can been seen, both are reduced when

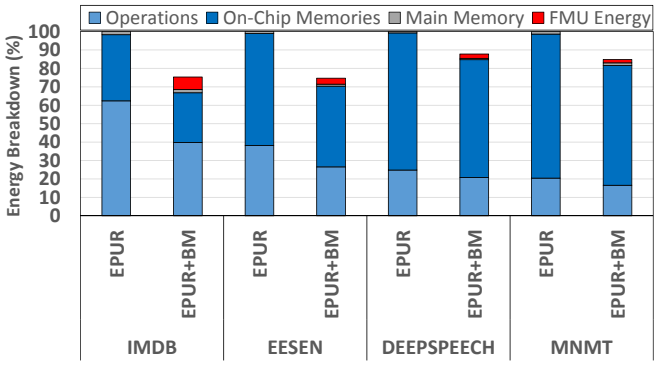


Figure 5.13: Energy breakdown for E-PUR and EPUR+BM.

using our memoization scheme. In E-PUR+BM, each time a value from the memoization buffer is reused, we avoid accessing all the neuron's weights and the input buffers, achieving significant energy savings. In addition, since the extra buffers used by E-PUR+BM are fairly small (i.e. 8 KB), the energy overhead due to the memoization scheme is less than 3% on average. The energy consumption due to the operations is also reduced, as the memoization scheme avoids neuron's computations. Furthermore, the leakage of scratch-pad and operations are also reduced due to the speedups achieved by the memoization scheme. Finally, the energy consumption due to accessing main memory is not affected by our technique since both, E-PUR and E-PUR+BM, must access main memory to load all the weights once for each input sequence.

Figure 5.14 shows the performance improvements for the different RNNs. On average, an speedup of 1.35x is obtained for a 1% accuracy loss, whereas accuracy losses of 2% and 3% achieve improvements of 1.5x and 1.67x respectively. The performance improvement comes from avoiding the dot product computations for the memoized neurons. Therefore, the larger the degree of computation reuse the bigger the performance improvement. Note that the memoization scheme introduces an overhead of 5 cycles per neuron (see Table 5.1), mainly due to the evaluation of the binarized neuron. In case the full precision neuron evaluation can be avoided, our scheme saves between 16 and 80 cycles depending on the RNN. Therefore, configurations with low degree of computation reuse, like Deepspeech at 1% accuracy loss, exhibit smaller speedups due to the overhead of the memoization scheme. On the other hand, RNNs that exhibit higher computation reuse, such as EESEN at 2% accuracy loss, achieve an speedup of 1.55x.

E-PUR has an area of 64.6 mm^2, whereas E-PUR+BM requires 66.8 mm^2 (4% area overhead). The largest overhead contribution (3%) is due to the extra scratch-pad memory required by the memoization unit.

5.5 Related Work

Increasing energy-efficiency and performance of RNN networks has attracted the attention of the architectural community in recent years [43, 62, 41, 61]. Most of these works employ pruning and compression techniques to improve performance and reduce energy consumption. Furthermore,

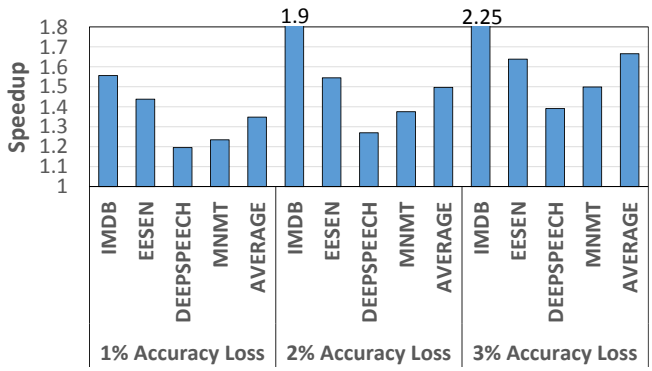


Figure 5.14: Speedup of E-PUR+BM over the baseline (E-PUR).

linear quantization is employed to decrease the memory footprint. On the contrary, our technique improves energy-efficiency by relying solely on computation reuse at the neuron level. To the best of our knowledge, this is the first work using a BNN as a predictor for a fuzzy memoization scheme. BNNs have been used previously [22, 80, 56] as standalone networks, whereas we employs BNNs in conjunction with the RNN to evaluate neurons on demand.

Fuzzy memoization has been extensively researched in the past and has been implemented both in hardware and software. Hardware schemes to reuse instructions have been proposed in [95, 5, 35, 12]. Alvarez et al. [6] presented a fuzzy memoization scheme to improve performance of floating point operations in multimedia applications. In their scheme floating point operations are memoized using a hash of the source operands, whereas in our technique a whole function (neuron inference) is memoized based on the values predicted by a BNN.

Finally, software schemes to memoize entire functions have been presented in the past [111, 3]. These schemes are tailored to general purpose programs whereas our scheme is solely focused in RNN , since it exploits the intrinsic error tolerance of RNNs.

5.6 Conclusions

In this chapter, we have shown that about 25% of neurons in an RNN change their output value by less than 10%. This motivated us to propose a fuzzy memoization scheme to save energy and time. A major challenge to perform neuron level fuzzy memoization is to predict, in a simple and accurate manner, whether the output of a given neuron will be similar to a previously computed and cached value. To this end, we propose to use a Binarized Neural Network (BNN) as a predictor, based on the observation that the fully precision output of a neuron is highly correlated with the output of the corresponding BNN. We show that a BNN predictor achieves 24.2% computation reuse on average, which is very similar to the results obtained with an Oracle predictor.

The experimental results show that our memoization scheme achieves significant time and energy savings with minimal impact in the accuracy of the RNNs. When compared with the E-PUR accelerator, our scheme achieves 18.5% energy savings on average, while providing 1.35x speedup at the expense of a minor accuracy loss.

6

Dynamic Precision Selection

In previous chapters, the performance and energy efficiency of RNN inference is improved either by changing the order of computations or avoiding them. In this chapter, we follow a different approach by focusing on the precision (bit-width) used to perform the calculations. More specifically, we explore the use of dynamic precision selection during RNN inference.

As mentioned earlier, RNN models tend to require a large amount of memory for storage. In this regard, linear quantization is one of the most popular and effective optimizations employed to reduce the storage requirements of RNN models. Linear quantization maps floating-point values to the product of an integer index and a quantization step, as detailed in Section 2.3.1. Therefore, the parameters and inputs are commonly stored as integer values, and the computations are performed using integer arithmetic [51, 19, 43]. For this reason, energy efficiency and performance are also significantly improved since it reduces the amount of energy required for computations and memory accesses.

Traditionally, when linear quantization is employed, the bit-width is set to the minimum precision that can retain the original model's accuracy based on an offline profiling, and it is kept constant during RNN inference. For instance, RNN accelerators such as E-PUR and TPU [51] use 8-bit weights and inputs for RNN inference.

Other proposals, such as Stripes [52] and Bit Fusion [92], support variable precision to further improve performance and energy efficiency for RNN layers that can be computed with less than 8 bits. Despite the additional flexibility of these accelerators, the bit-width for each RNN layer is determined offline, and it is fixed during inference. In other words, different RNN layers can be evaluated at different precision, but a given layer is always computed at the same bit-width for all the inputs. In this chapter, we propose a mechanism to dynamically select the precision used during the evaluation of a neuron aiming to boost performance without any accuracy loss.

A major challenge to select the precision dynamically is deciding when to change precision. We observe that for RNNs, the cell state can be useful as it stores information from previous inputs that will be used for future predictions. Therefore, we investigate a practical scheme to set the bit-width online based on the cell state.

In the rest of this chapter, we go over the motivations for employing dynamic precision. Then, we present an analysis of the importance of the cell state when deciding the precision to be used. In the later sections, we describe our proposal for dynamic precision selection. Finally, we detail our experimental results and conclusions.

6.1 Benefits of Dynamic Precision Selection

RNN cells are composed of four gates, each one of them with two matrices containing the weights for the forward and the recurrent connections, respectively. Since these weight matrices tend to be quite large, most of the energy consumed by state-of-the-art hardware accelerators for RNN inference is due to the static and dynamic energy consumed by the memories employed to store the weights and intermediate results.

An effective way to decrease memory footprint and thus static and dynamic energy without affecting accuracy is using Linear Quantization (see Chapter 2). Typically, a static profiling of the network is done to determine the minimum precision that can be used to quantize an RNN model without losing accuracy. A common approach is to set a fix bit-width (i.e., 8 bits) for the whole network. However, while this solution covers the worst case, it ignores cases where a lower precision could be employed for a subset of computations without losing accuracy.

Figure 6.1 shows the accuracy loss of an RNN model for speech recognition [69] when using 8 bits, 4 bits, and a mix of both. In the case of the mixed-precision, for each neuron (e.g., inputs and weights), we dynamically set the precision to 8 or 4 bits as described later in Section 6.3.1. As shown in Figure 6.1, using a precision of 8 bits (i.e., assuming worst-case bit-width for the whole network) results in no accuracy loss. On the contrary, using a precision of 4 bits incurs in 2% of accuracy loss (which is an important loss for speech recognition). However, it can be observed that more than 50% of the computations can be evaluated using 4 bits while the rest are computed using 8 bits without losing accuracy.

Previous works have reported and exploited variability in the precision requirements for DNN computations [52, 53]. However, they exploited precision variability across layers, whereas in this work, we focus on precision variability among neurons and time-steps of execution, which is a much finer-grain variability. In other words, prior proposals support different precision for different DNNs, but the precision for each DNN and layer is determined offline and kept constant during inference. Our proposal is different as we dynamically select the precision for each neuron and time-step.

Regarding the benefits of dynamic precision selection, they are shown in Figure 6.1. Employing 4-bits precision to evaluate all time-steps reduces the execution time to 50% of the 8-bit version. However, it would introduce a significant degradation to the model accuracy. Our scheme, illus-

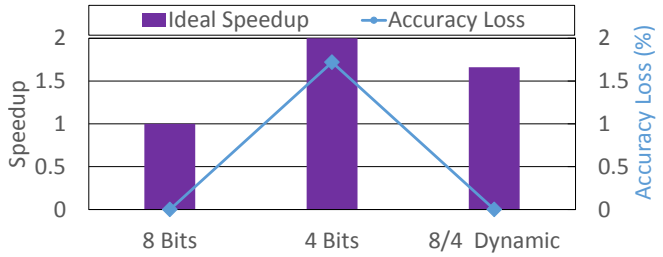


Figure 6.1: Speedup and accuracy loss for a speech recognition LSTM network [69] using different schemes to set the precision. 8 Bits and 4 Bits are configurations that fix the precision to 8 and 4 bits respectively for all the time-steps. 8/4 Dynamic is our scheme for dynamically selecting the precision at run-time, which employs 4 bits for stable regions of the cell state and 8 bits for peak regions. As it can be seen, our dynamic scheme outperforms the 8 bits version without any accuracy loss.

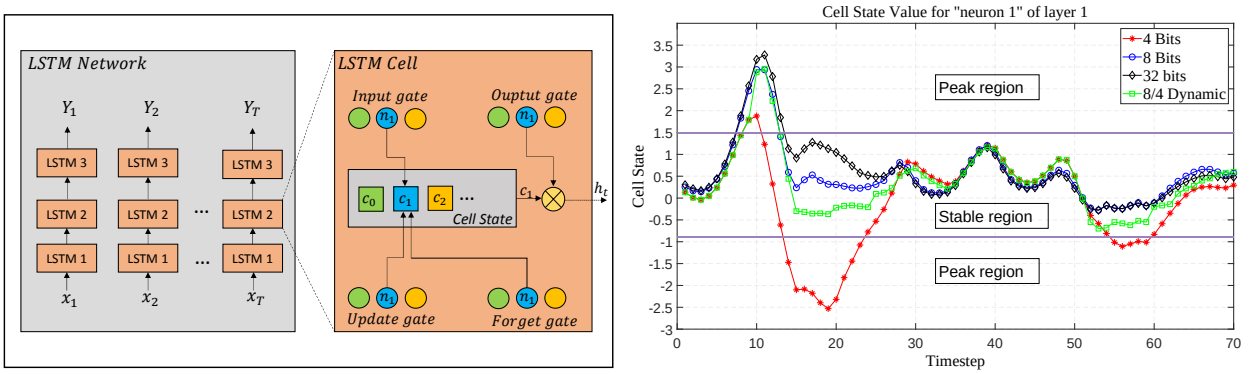
trated in the second bar, restricts the use of 4-bit quantization to certain regions of the cell state, representing around 66% of the time. In contrast, the remaining regions are computed using 8 bits. By doing so, we leverage 4-bit quantization for a large percentage of the execution, while avoiding any accuracy loss.

A primary challenge to dynamically change the precision is deciding when to use a high or low precision. In this work, we propose to use the state of the cell state as an indicator of the required precision. We describe the rationality behind this decision in the next section.

6.2 Importance of the Cell State

The cell state is a critical component of an RNN cell as it stores the cell information. As described in Section 2.2.3, it consists of an array of N elements, where several neurons from different gates are employed to compute each element, i.e. the gates behave as fully-connected layers. Furthermore, we use the term cell state to refer to the hidden state (h_t) of GRU cells and the cell state (c_t) of LSTM cells.

Figure 6.2b shows the evolution of one element in the cell state in a speech recognition network [69], at three different levels of precision (32-bit floating-point, 8-bit integer, and 4-bit integer). As it can be seen, 8-bit quantization closely tracks the behavior of the 32-bit full-precision version, resulting in the same accuracy. However, 4-bit quantization introduces significant errors in some time-steps, resulting in noticeable accuracy loss. Previous schemes would conclude that this LSTM network layer cannot be evaluated using 4 bits. However, a more detailed look at Figure 6.2 reveals that the 4-bit version can mimic the behavior of the 32-bit version for a large percentage of time-steps. More specifically, for phases where the cell state is stable the 4-bit version is quite accurate, whereas for phases where the cell state changes rapidly, i.e., peaks/valleys, it tends to exhibit a larger error. A more extensive analysis by using different LSTM networks and their respective training datasets shows that this behavior is quite prevalent. For stable phases, the 4-bit version introduces an error of 19.6%. On the other hand, for peaks/valleys, it introduces a larger error of



(a) Cell state vector update

(b) Cell state for element n_1

Figure 6.2: Evolution of one element (n_1) in the cell state of a speech recognition LSTM network [69]. As shown in (a), on each time-step, the elements in the cell state vector are updated using a combination of the gates' output and the previous cell output. Also, for each LSTM layer, the cell state is different, and the last layer's output on each time-step is used to generate the output sequence. Shown in (b) is the cell state for element n_1 of the cell state, i.e. it plots the value of cell state element c_1 for different precision levels. As it can be seen, in stable regions the low precision (4-bit version) evaluation accurately tracks the behavior of the high precision (FP32) version. However, a large error is introduced when the tracked element is on a peak.

78% on average. For brevity, we will use the term peak to refer to both peaks and valleys.

Based on this observation, we propose a scheme that dynamically selects the appropriate precision by monitoring the cell state of the RNN cell. Our system keeps track of the values of each element in the cell state in recent time-steps. If the value is inside a stable region, the lowest precision supported by the hardware is selected to evaluate the next time-step. Otherwise, higher precision is used (8 bits) to avoid significant errors during the peak regions. In our set of RNN models, this simple scheme allows us to use the lowest precision for more than 56.2% of the time without any accuracy loss. Note that our scheme dynamically changes the precision for all the elements of the cell state individually. Consequently, at each time-step, some neurons are evaluated using high precision, whereas the rest are evaluated using low precision. As shown in Figure 6.2b, the value of the cell state when applying our scheme (labeled 8/4 dynamic in the figure) follows the cell state of the 32-bits version closely. Note that, in the peak regions, the error of *8/4 dynamic* is smaller than the error of the 4-bits version.

In summary, we design a scheme that tracks the evolution of the cell state at run-time. Then, for each of its elements, it selects a high precision during the peaks and a low precision for stable regions. For this work, we use 8 bits for the high precision since it provides zero accuracy loss for all tested RNN models. On the other hand, we use 4 bits for the lower precision, which would have a significant loss in accuracy if it was used for all the time-steps.

Regarding the range of values of the weights, we have noted the presence of outliers, as previously explored in [75]. As in [75], by quantizing the outliers using high precision (i.e., 8 bits), the remaining values could be quantized more aggressively (i.e., using lower precision) without affecting

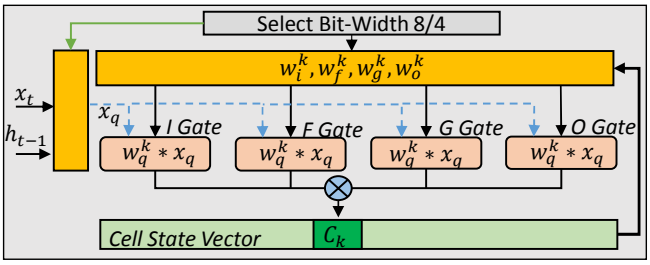


Figure 6.3: Relationship among neurons in the four gates and elements in the LSTM cell state. The value Ct_k of element n_k is computed based on the outputs of the k^{th} neuron in each of the four gates. The precision used to evaluate those neurons is based on the evolution of the element n_k. Based on the selected bit width, x_t and w^k are quantized to x_q and w_q^k, respectively.

the overall accuracy of the network. Therefore, in this work, we follow a similar approach where outliers are quantized using higher precision than the rest of the values.

In the following sections, we detail this scheme and describe its hardware implementation on top of E-PUR. For brevity, we based our explanations on an LSTM cell. However, the same principles apply to GRU cells.

6.3 Dynamic Precision Scheme

6.3.1 Overview

Our principal goal is to set the precision at each time-step of execution for the input vectors x_t and h_{t-1} and their corresponding weights for each single element of the cell state individually. For a given LSTM cell, the k^{th} element of the cell state vector is computed using a combination of the output value of the k^{th} neuron on each gate. We refer to these four neurons simply as element n_k of the LSTM cell and set the precision for the four of them in tandem, since all of them are associated with the same element of the cell state. This relationship is shown in Figure 6.3.

To determine when each element n_k of the cell state is on a peak, we employ the state machine depicted in Figure 6.4. To track the evolution of the value of the cell state (c_k) of a given element n_k, we divide the process into three phases. First, the system starts in a *profiling state* that samples c_k for a certain number of time-steps. This profiling is done in order to determine the peak characteristics of c_k. Then, we have the *stable state* that indicates that c_k has had a stable value for the previously evaluated time-steps. Finally, the *in-a-peak state* tracks when c_k is at a peak.

As shown in Figure 6.4, the *profiling state* is performed for T time-steps. In each profiling step, we keep track of the maximum and minimum value of the cell state. Note that the profiling is done using low precision (4 bits) because we assume that while profiling the cell state is inside a stable region. Finally, after T time-steps, we use the maximum and minimum value of c_k to set the limit values that define when a peak begins or ends, and then we move to the *stable state*.

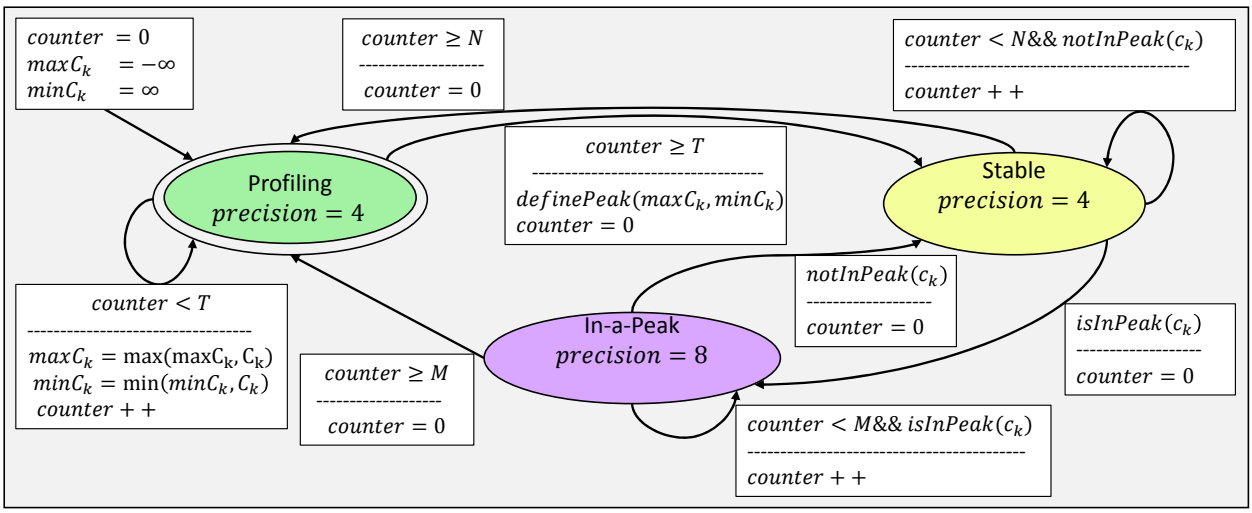


Figure 6.4: State machine employed to dynamically select the precision for an element c_k of the cell state.

$$r = maxC_k - minC_k \tag{6.1}$$

$$upperLimit = r + r * \beta \tag{6.2}$$

$$lowerLimit = r - r * \beta \tag{6.3}$$

$$isInPeak = lowerLimit \leq c_k \leq upperLimit \tag{6.4}$$

Figure 6.5: Positive and negative peak region definition for the cell state of a given neuron (i.e., n_k).

The system remains in the *stable state* until a peak is detected. A peak is found using the values $minC_k$ and $maxC_k$, obtained previously in the profiling state, as shown in Figure 6.5. To determine that the value in the cell state has entered a peak, we require that it exceeds the $minC_k$ and $maxC_k$ found during the profiling stage by a given margin to increase the confidence of the detection. To this end, we use the parameter β in Equation 6.2 and Equation 6.3 to establish the upper and lower thresholds. If the c_k value in the cell state exceeds one of these thresholds, a peak is detected, and the system transitions to the *in-peak-state* to use high precision. It remains in this state until we detect that c_k is no longer in a peak using Equation 6.4. In case that the end of the peak is detected, we move to the *stable state* to switch back to low precision, as the value of the cell state has entered a stable phase.

If the system stays in a peak for a large number of time-steps (e.g., M in Figure 6.4), the profiling stage is triggered again. Note that this profiling is needed since the value of c_k may

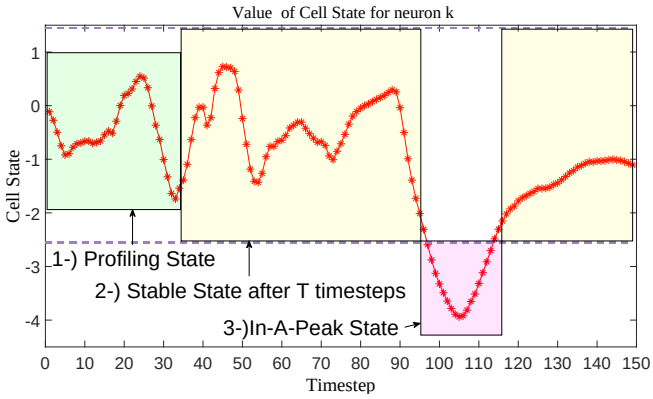


Figure 6.6: LSTM cell state's evolution for a given neuron on multiple time-steps. At each time-step, its stability is tracked to decide the precision for the next time-step. Dash lines (- -) represents the lower and upper limit that defines the stable and peak region after the profiling is completed.

become stable at a value outside the thresholds of the original profiling. In this case, our scheme would stay indefinitely in the high precision state if the profiling is not repeated to set new upper and lower thresholds. In other words, the initial profiling information may become outdated since the range of values of the cell state may shift over time. On the other hand, the system may be stuck in the *stable state* in case the range of the values of the cell state becomes narrower over time, as they will never exceed the minimum and maximum thresholds set in the initial profiling. To prevent this issue, we force a profiling stage when the system stays in the *stable state* for more than N time-steps. By doing this, we take into consideration the newest values of the cell state, and more adequate thresholds are set.

The overall scheme for dynamic precision selection is summarized in Figure 6.6. Considering an input sequence with elements x_0 to x_{n-1}, for a given cell state element n_k, the scheme works as follows. First, the value c_k of the element n_k is computed using low precision and the *profiling state* is executed for T time-steps, marked as 1) in Figure 6.6. Then, after the profiling stage, the system moves to the *stable state* and performs all computations associated with n_k using low precision. Then, at time-step p, we detect that c_k is lower than its previously profiled lower threshold and, thus, the system changes to the *in-a-peak state*, as seen in Figure 6.6. Next, it stays in this state until the value of c_k comes back to its previously profiled range and then switches back to *stable state*, where it waits for the occurrence of another peak or the triggering of another profiling stage. Note that this process is performed for each element in the cell state individually and, hence, our system may select different precisions for different neurons in the same time-step.

6.4 Hardware Support for Dynamic Precision

We implement this dynamic precision selection scheme on top of E-PUR. As described in Chapter 4, E-PUR consists of several computational units that evaluate the four different gates in an LSTM cell. Furthermore, it includes on-chip storage for weights and intermediate results. In E-PUR, computations are done using 8-bit parallel multipliers. To support variable precision, we

replace the 8-bit parallel multipliers by multi-precision multipliers: either multi-precision parallel multipliers or SIP units. We describe the multipliers employed to evaluate our proposal and explain the architectural extensions added to E-PUR to implement our scheme for dynamic precision selection in the next subsections.

6.4.1 Multi-Precision Multipliers

Multi-precision multipliers are commonly employed on works that use mixed-precision for their computations. These multipliers perform multiplications in parallel [92] or serially [52]. Regarding multi-precision parallel multipliers, they can be configured for a low precision (i.e., 4-bits) or a high precision (i.e., 8-bits) mode. When operating on low precision mode, they either work on half of the cycles needed to complete the high-precision multiplication, or they can provide twice the throughput.

Regarding Serial Inner Products units (SIPs), they have been previously proposed and used [52] as a mechanism to exploit precision variability in different layers of a neural network. In a SIP unit, an inner product is computed by serially feeding the bits of one of the operands while the bits of the other are fed-in parallel. On a cycle, a SIP unit performs the element-wise multiplications between a vector with *1-bit* elements and a vector with *n-bits* elements. These multiplications are typically done using AND and SHIFT operations. Then, the summations are performed by accumulating the partial products computed on each cycle. Note that only one bit from each element of one of the operands is multiplied on a given cycle. Thus, decreasing the bit width of the elements in the vector that is being fed serially will result in a linear increase for the SIP performance (i.e., reducing the bit width from 8 to 4 will result in 2x speedup).

One advantage of SIP units over multi-precision parallel multipliers is that they allow a finer granularity when setting the precision. Nonetheless, our proposal is independent of either multi-precision parallel multipliers or SIP units.

6.4.2 Hardware Baseline

We extended E-PUR to use multi-Precision multipliers instead of parallel multipliers to compute inner products. Figure 6.7 shows the structure of a computational unit (CU), which is tailored to the evaluation of a gate in an RNN cell. A CU is composed of a dot product unit (DPU), a Multi-Functional Unit (MU), and several buffers to store the weights and inputs. DPUs are used to compute the matrix-vector multiplications in the four gates.

MUs are used to evaluate activation functions and scalar operations using floating-point numbers. They are also used to quantize the cell output (h_{t-1}) and to convert the DPU output to a floating-point. Note that weights are quantized offline.

As described in Chapter 4, in E-PUR, all the gates are evaluated in parallel using a fixed-precision of 8-bits. For a given time-step (x_t) of an input sequence (X), the following steps are performed to compute the output vector (h_t). First, for each output element (i.e, n_k) of h_t, the input and weight vectors are split into K sub-vectors of size N. Then, two sub-vectors of size N

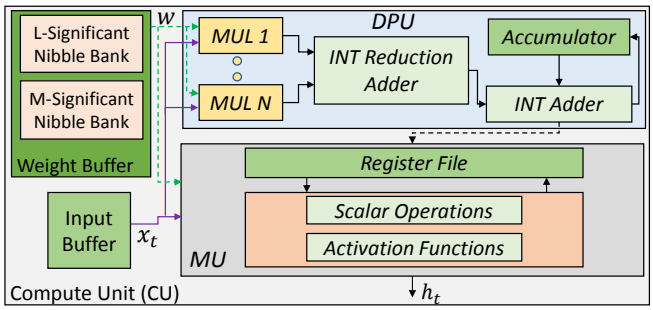


Figure 6.7: Compute Unit (CU). Multi-precision multipliers or SIP units are included to support variable precision.

are loaded from the input and weight buffers, respectively, and the dot product between them is computed by the DPU, which also accumulates the result. Next, the steps are repeated for the next k^{th} sub-vector, and its result is added to the previously accumulated partial dot product. This process is repeated until all K sub-vectors are computed and added together.

After a DPU computes its output value (y_t), it is sent to the MU, where it is converted to floating-point, and the activation function is computed. After each gate is evaluated, the cell state is computed and stored in the input buffer by the output gate. Also, the output value (h_t) is computed and quantized. Finally, the MU stores the final result in the on-chip memory for intermediate results. Note that the operations to compute the dot product, the activation function and to quantize the result are overlapped, as seen in Figure 6.8. Hence, once the DPU sends a result to the MU, it will continue with the next neuron. Similarly, once an activation function is computed, we proceed with the computation of the subsequent activation while applying the quantization steps to the previously computed activation. These steps are repeated until all the neurons in the LSTM cell are evaluated.

Regarding weights that are outliers, they are quantized using 8-bits. Their indexes and pointers to their corresponding input element are stored in a buffer (*outlier buffer*). Furthermore, a zero is stored in the weight buffer for each weight that is an outlier. It is done to efficiently fetch all the weights without having to include an index to their corresponding input. Finally, their computation is performed after completing the evaluation of the non-outliers. Note that processing the outliers in this manner represents a minimal overhead, since less than 1% of the weights are considered outliers in our set of benchmarks.

Mixed Precision with Multi-precision parallel multipliers

To support different precision with parallel multipliers, we change the 8-bit parallel multipliers in the baseline implementation of E-PUR by multi-precision parallel multipliers (MPP). More specifically, we employ MPPs that operate in two precision modes high (i.e., 8-bits) and low (i.e., 4-bits). When working in high precision mode, an LSTM cell is evaluated as explained in the previous section. However, two multiplications are done per multiplier on each cycle when operating in low precision mode. Therefore, for each multiplier, we double the number of operands that are fed to them. Since the precision is lower (i.e., half), the bandwidth requirements do not increase.

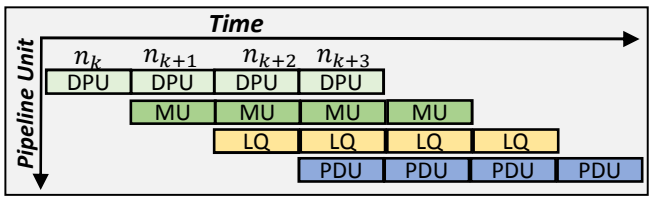


Figure 6.8: Overlapping of computations in the accelerator.

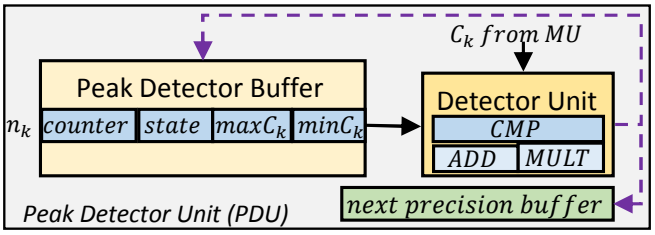


Figure 6.9: Structure of the Peak Detector Unit (PDU).

Mixed Precision with SIP

To support mixed-precision using SIP units, the steps described in Section 6.4.2 are performed as follows. First, to maintain the same throughput as the baseline, 8 SIP units are included per DPU. Then, on each CU, eight sub-vectors of weights with N elements are fetched from the weight buffer and dispatched to each SIP unit. Furthermore, an $8*N$-bit vector (v_0), corresponding to the most significant bit of each element in x_t, is fetched from the input buffer and dispatched to each SIP unit to perform the multiplication of v_0 with the corresponding weights. After this, each SIP accumulates its output. Next, this process is repeated until all the bits in x_t are multiplied and added together. Finally, the accumulated values on each SIP unit are added together, and the process is repeated for the remaining sub-vectors.

6.4.3 Supporting Variable Precision

To set the precision for each neuron at each time-step, we extend EPUR with a Peak Detector Unit (PDU), as shown in Figure 6.9. This unit tracks the evolution of each element in the LSTM cell state. As seen in Figure 6.9, the PDU is composed of a buffer that stores the information needed by the state machine shown in Figure 6.4. Furthermore, it includes a detector unit that is employed to detect when the tracked cell state is inside or outside a peak, according to Equation 6.4. The PDU updates the state for a given element of the cell state after the MU computes its value and sets the precision to be used in the next time-step in the *next precision buffer*. Note that the computations performed by the PDU are overlapped with the MU evaluations in a pipelined manner, as shown in Figure 6.8.

One major challenge to dynamically set the precision is storing the quantized integer indices for the weight matrix and input vectors in an energy-efficient manner. This challenge arises because, at each time-step, an index can be fetched in either low or high precision. Therefore, a mechanism

that can store and fetch both indices in an energy-efficient way is needed. One possible solution is that for a given floating-point value, the index for high and low precision are stored separately. The main drawback for this approach is that the memory footprint increases by 50%, resulting in a significant increase in energy consumption of the overall system.

A more energy-efficient alternative is to use only one byte to store the high and low precision indices for a single weight. In this approach, for a given floating-point value, which is quantized in low and high precision, the most significant nibble of its high precision index can also be used as a low precision index. Therefore, the memory footprint of the baseline system is not increased. Furthermore, if the most and least significant nibbles of a given index are stored separately, only half of the memory accesses are required to fetch the low precision indices, hence dramatically decreasing the dynamic energy consumption of the system.

One drawback of using the most significant nibble of a high precision index as low precision index is that they are not always equal. Figure 6.10 shows a mapping of some floating-point numbers to integer indices using 8 bits (top of the figure) and 4 bits (bottom of Figure 6.10). As it can be seen, using just the most significant nibble of the 8-bit index to obtain the 4-bit index is incorrect for half of the cases. As an example, a floating-point value mapping to index 11 using 8 bits is quantized as 1 when using 4 bits. However, using the most significant nibble of the 8-bit index would give an incorrect mapping of 0. A key observation is that values that are incorrectly mapped to its low precision counterpart always have a difference of one with the correct representation. Therefore, the right index can be obtained by adding one to the most significant nibble of the corresponding high precision index. Note that only the upper half of the high precision indices (index 8 to 15) have an incorrect mapping. For simplicity, we only show in Figure 6.10 the first 16 indices and negatives values are omitted. However, the same issue would arise for the rest of the indexes.

In this work, after a given weight is quantized for low and high precision, the low precision index and the least significant nibble of the high precision index are stored in memory. Then, we use the low precision index to obtain the most significant nibble of its high precision counterparts by either using it directly or subtracting one from it when an incorrect mapping is found. As an example, in Figure 6.10, a floating-point value mapped to index 11 using 8 bits and to 1 when using 4 bits will be stored in memory as 0x1B. Then, when the low precision index is needed, we fetch the most significant nibble (1 for this example). However, if the high precision index is needed, the whole byte is fetched (0x1B). In this case, we have an incorrect mapping. Thus, we subtract one from the most significant nibble. Note that we could detect an incorrect mapping by checking if the least significant nibble is in the upper half (greater than 7). We included in the accelerator the extra hardware required to detect and correct incorrect mappings. Since weights are static, they are encoded offline. Regarding the input elements, their mapping is done online. Finally, we store the low precision indexes and the least significant nibble of the high precision indexes in different memory banks. Specifically, we employ the *l-significant nibble bank* and the *m-significant nibble bank* to store the low and high precision indices, respectively, as shown in Figure 6.7.

To evaluate an LSTM cell, in addition to the steps described in Section 6.4.2, we also perform some extra tasks to set the appropriate precision for each neuron. First, for a given neuron n_i, the bit-width to be used is read from the next precision buffer in the PDU. Then, if high precision is chosen, we fetch the bytes corresponding to each element in the weight vector of n_i. Then, for each

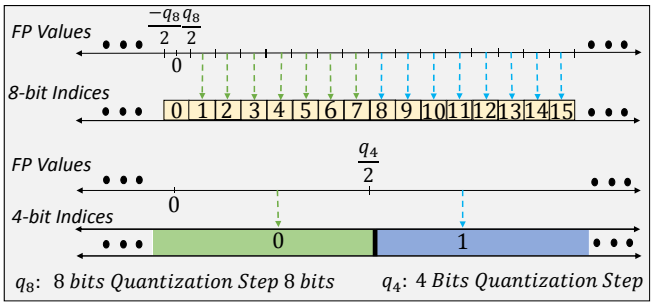


Figure 6.10: Linear Quantization of floating-point values for 8 and 4 bits. In some cases, using the most significant nibble of an 8-bit index to represent the corresponding 4-bit index yields an incorrect mapping.

byte fetched, we use the least significant nibble to detect if the most significant nibble is correct or not. For incorrect cases, the most significant nibble is adjusted by decreasing it by one unit. On the other hand, if low precision is selected, we only fetch the most significant nibble. Once the values have been fetched and adjusted, they are sent to each of the DPU as outlined in Section 6.4.2.

Regarding the input vector, we first fetch all its elements in high precision and low precision since the same input vector is used for all the neurons. Then, we proceed to feed them serially to each DPU depending on the selected precision.

Finally, once we compute the cell state in the MU, it is sent to the PDU to determine the precision to be used on element n_i in the next time-step. In addition, the output value is sent to the quantization unit, where it is quantized. Note that these operations are overlapped, as shown in Figure 6.8, and their latency is hidden by the computations in the DPU.

6.5 Experimental Results

This section presents the evaluation of the proposed technique to dynamically select the precision based on the stability of the LSTM cell state. We employ the methodology presented in Chapter 3. Moreover, we use some of the networks shown in Table 3.4 as benchmarks. The original accuracy for each network, using 32-bit floating-point arithmetic, is reported in Table 3.4.

The state machine for dynamic precision selection, shown in Figure 6.4, requires three different parameters: M and N control the maximum number of time-steps that the system may remain in states *In-a-Peak* and *Stable* respectively before triggering the profiling, whereas β is used to set the upper and lower thresholds to decide whether the value of the cell state is inside a peak. We performed a design space exploration for these parameters and found values that provide good results across the four LSTM networks used. Therefore, these parameters do not have to be manually tuned for each new LSTM network, as we empirically determined that the values shown in Table 6.1 provide excellent results for a wide variety of networks. Furthermore, to prove that these values generalize well for new unseen inputs, we performed the design space exploration by using the training datasets, whereas the evaluation of the technique is performed by using the test

Table 6.1: Hardware Configuration.

E-PUR	
Parameter	**Value**
Technology	28 nm
Frequency	500 MHz
Intermediate Memory	6 MiB
Weight Buffer	2 MiB per CU
Input Buffer	8 KiB per CU
DPU Width	16 operations
MU Latency	20 cycles
LQ Latency	8 cycles
MU Communication	2 cycles
Outlier Buffer	128KiB
Peak Detector Buffer	8 KiB
State Machine Configuration	
M	5% of time-steps
N	5% of time-steps
β	0.1

datasets.

For our experiments, the baseline system is E-PUR using 8-bit parallel multipliers, labeled as E-PUR+PAR. We evaluate our scheme on top of E-PUR using SIP and parallel multi-precision multipliers. The system implementing our dynamic precision selection scheme and SIP units is labeled as SIP+DYN. In contrast, the system with the multi-precision multipliers and our technique is marked as PAR+DYN.

The rest of this section is organized as follows. First, we present an evaluation of the effectiveness of our scheme. Second, we provide the performance and energy results. Third, we evaluate our scheme using three levels of precision (8, 4, 2). Finally, we analyze the area overheads of the proposed scheme.

Figure 6.11 reports the effectiveness of using the cell state stability to set the precision dynamically. In this figure, we compare our proposal with a scheme that randomly selects the precision level for each element of the cell state at each time-step. The random scheme has a low precision usage of 34% on average, i.e., 34% of the evaluations are performed at low precision (4 bits) whereas 66% are done at high precision (8 bits). However, the random scheme produces a significant loss in accuracy for all the networks. On the other hand, our scheme has a 49% low precision usage on average, without any accuracy loss. Therefore, tracking the stability of the LSTM cell state provides valuable information to select the precision. Note that the IMDB model can be evaluated using only 4-bits for both approaches.

Figure 6.12 shows the performance improvements for our set of RNN networks. On average, a speedup of 1.46x is obtained without any accuracy loss. SIP+DYN and PAR+DYN exhibit the same speedup since both of them have the same throughput and they operate at the same frequency.

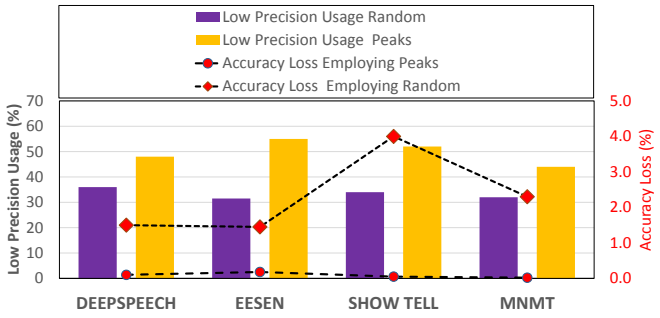


Figure 6.11: Comparison between our scheme ("Peaks") and a system that randomly chooses the evaluations done at low precision ("Random"). The random scheme produces a significant degradation in accuracy. Our scheme achieves higher coverage without any accuracy loss.

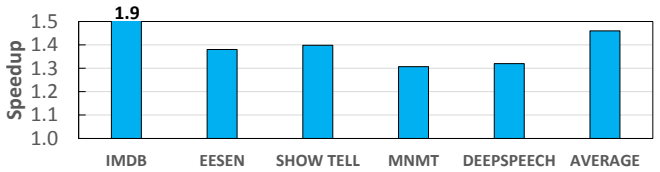


Figure 6.12: Speedups achieved by changing the precision dynamically. Baseline configuration is E-PUR with 8-bit parallel multipliers.

As seen in Figure 6.12, all the models achieve consistent and significant speedups when compared with the baseline. The reduction in execution time is due to using lower precision (4-bit) for more than 57% of the time. Note that the baseline employs 8 bits for all the computations to maintain the accuracy. Furthermore, the smaller the bit-width the higher the performance of the E-PUR+DYN and E-PUR+SIP: switching from 8 bits to 4 bits doubles the performance of the dot product units. Hence, for time-steps and cell state elements where 4-bit precision is used (stable regions), the latency of the dot product is reduced by a factor of 2x compared to the 8-bit version. This represents around 60% of the evaluations for our set of RNN networks on average. On the contrary, 40% of the evaluations are still done using 8 bits to maintain accuracy (peaks regions of the cell state). Thus, their performance is not improved.

The obtained speedup is close to the theoretical performance improvement. The foremost reason is that our scheme's latency is largely hidden due to overlapping the DPU and multi-functional unit computations. Also, all the CUs are fully utilized. Note that most of the execution time is due to the dot product calculations to evaluate each *neuron* on each gate using either 4 or 8 bits. In both cases, the latency of the dot product calculations is larger than the latency to compute the activation functions in the MU and to quantize the output in the LQ unit.

For all the networks, just a small execution time overhead is added when the evaluation of an LSTM cell starts (i.e., first neuron and first time-step). The IMDB network has a speedup of 1.99x since it can be evaluated entirely in low precision, and our scheme can detect it. For this model, we do not observe any peak regions in the cell state, and thus, low precision is used for all the evaluations. The networks EESEN, SHOWTELL, and NMT achieve a speedup of 1.38x, 1.40x, and 1.31x, respectively, since they require high precision for some of their evaluations. Regarding

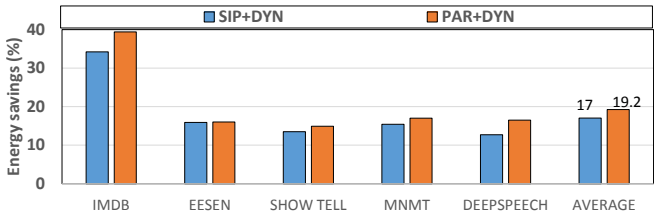


Figure 6.13: Energy savings achieved by dynamically changing the precision compared to the baseline (E-PUR-PAR). SIP+DYN and PAR+DYN refers to our scheme using SIP units and multi-precision parallel multipliers, respectively.

the DeepSpeech network, a speedup of 1.32x is achieved.

Figure 6.13 shows the energy savings achieved by PAR+DYN and SIP+DYN when compared to the 8-bit baseline, including both static and dynamic energy. On average, the savings in energy consumption for PAR+DYN and SIP+DYN are 19.2% and 17%, respectively. These savings come from several sources. First, using lower precision (4 bits) reduces the dynamic energy of the weight buffer, since less amount of information is fetched compared to the 8-bit version. More specifically, when using low precision, we only fetch the least significant nibble of the weights and inputs. Second, the energy cost of computing the dot product is reduced when employing 4 bits instead of 8 bits, since the activity in the DPUs is reduced. Finally, the speedups reported in Figure 6.12 provide a reduction in static energy.

As it can be seen in Figure 6.13, using our scheme with parallel multipliers (DYN-PAR) provides 2% higher energy savings than the SIP implementation. The reason for this is that the SIP unit includes extra components to perform shifting and accumulation of the products computed on each cycle. However, as mentioned previously, SIP units provide finer granularity than the parallel multipliers, being able to perform the dot product at any number of bits.

The LSTM networks EESEN and IMDB exhibit the most substantial energy savings: 15.9% and 34.2%, respectively, when using SIP+DYN. For PAR+DYN, their savings are 16% and 39.4%. For these networks, a large percentage of the computations are evaluated using low precision and, thus, the energy savings are significant. Most of the energy savings for these two networks are due to the reduction in static and dynamic energy of the scratchpad memories, which are used to store the weights. For the networks SHOWTELL and NMT evaluated on SIP+DYN, the energy savings are 13.5% and 15.4%, respectively, whereas when evaluated on PAR+DYN, their energy savings are 14.9% and 17%. These networks are the largest in our set of benchmarks and, hence, their dynamic and static energy consumption is larger than the other two networks. Regarding DeepSpeech, the energy savings are 12.7% and 16.5% for SIP+DYN and PAR+DYN, respectively.

Our scheme can be extended to support more than two levels of precision. To assess this, we evaluated EESEN and DeepSpeech using 8, 4 or 2 bits. A smaller range inside the stable region (shown in Figure 6.4) is defined, such that values of the cell state inside this new range are computed using 2 bits, otherwise they are done using 4 bits. For this scheme, 8.5% of the EESEN computations are performed using 2-bits, whereas, for DeepSpeech, 10% are done with 2-bits. As a result, the energy consumption and execution time of these networks are improved by 3.3% and 1.02x, on average.

Regarding the area, the E-PUR baseline has an area of 31.3 mm^2. SIP+DYN has an area of 33.6 mm^2, whereas PAR+DYN has an area of 33.2 mm^2. Therefore, a small overhead of around 8% is added by the buffers and multipliers required to support setting the precision dynamically.

6.6 Related Work

The Tensor Processing Unit (TPU) [51] is an ASIC that supports convolutional, fully-connected, and LSTM, delivering performance per watt orders of magnitude higher than CPUs and GPUs. It achieves a performance of 92 TOps/s (8-bit) while dissipating 40 Watts. TPU employs a fixed precision of 8 bits for weights and inputs. Our proposal is different as it selects the precision dynamically at runtime, using 4 bits for more than 66% of the time without any accuracy loss.

On the other hand, more flexible accelerators that support variable precision have been introduced in recent years. Stripes [52] uses a Serial Inner Product (SIP) unit in which the bits are fed serially, and the bit-width of the operands can be changed online. Stripes only accelerates convolutional and fully-connected networks, whereas LSTMs are the main focus of this work. BitFusion [92] is a bit-flexible accelerator that includes an array of bit-level processing elements that can be dynamically merged or split to match the bit-width of individual DNN layers. Also, it provides full support for LSTM networks. Although Stripes and BitFusion offer a significant degree of flexibility, the precision for each layer is determined offline and kept constant during inference, i.e., a layer always employs the same precision. In this work, we show that higher performance can be achieved by dynamically selecting the precision based on the cell state's evolution. Our scheme can change the precision for every element of the cell state and every time-step, further improving performance and energy efficiency.

In addition to the ASIC-based solutions, FPGA-based accelerators for LSTM inference have also been proposed in recent years. Brainwave [28] is a Neural Processing Unit that achieves an order of magnitude improvement in latency and throughput over state-of-the-art GPUs on large LSTM networks. The Efficient Speech Engine (ESE) [43] exploits pruning and sparsity to improve the performance of LSTM networks on FPGAs. C-LSTM [107] leverages structured compression techniques that reduce the LSTM model size while eliminating the irregularities of computation and memory accesses. On the other hand, DeltaRNN [31] and the work in [82] exploit temporal coherency of the LSTM data to reuse computations and avoid redundant memory accesses. Pruning, compression, and computation reuse techniques are completely orthogonal to our scheme.

Regarding software-based RNN quantization, some proposals such as HitNet [106] and Binary Forget and Input gate (BFIG) [63] have been introduced. These schemes apply quantization statically. Hitnet goes as low as 2 bits, whereas BFIG uses 1 bit to represent the forget and input gates. In the case of HitNet, it is only tested on a small model (1 layer and 300 units). We have seen in our experiments that small LSTM models tend to work better when employing low precision, whereas models such as NMT tend to lose accuracy when it is quantized with less than 4 bits. In the case of BFIG, accuracy is lost for the NMT network, whereas our scheme does not lose any accuracy. Also, it only works in two of the gates, whereas our scheme can employ low precision for the four gates. Moreover, as shown in the Section 6.5 we can go as low as 2 or 1 bit. Besides, if we are more aggressive while applying our scheme (allow accuracy loss and multi-level precision), our

scheme can be as low as 2 or 1.

6.7 Conclusions

In this chapter, we present a novel scheme to select the precision for RNN computations dynamically at runtime. We observe that the values of the cell state can be used to determine the required bit-width: time-steps where the value changes rapidly (i.e., peaks) require higher precision to avoid large errors, whereas time-steps where the value is relatively stable can be evaluated with lower precision. Based on this observation, we propose a novel scheme that monitors recent values of the LSTM cell state and sets the appropriate precision dynamically. Unlike previous schemes that fix the precision for each DNN layer offline, our system can change the precision for every cell state element and every time-step. We evaluate our proposal on top of E-PUR using four popular LSTM networks. The experimental results show that our scheme selects the lowest precision for more than 57% of the time without any loss in accuracy, providing 1.46x speedup and 19.2% energy savings on average. The extra hardware required for our technique is quite modest, as it represents a small area overhead of 8%.

7

Energy-Efficient and High-Throughput RNN Batching

In this chapter, we address the problem of performing RNN inference in an energy-efficient manner while providing high-throughput. First, we describe the importance of batching for RNNs and review the state-of-the-art approaches for RNN batching. Second, we analyze the primary sources of inefficiencies of current batching systems. Third, we present E-Batch, an energy-efficient batching scheme tailored to RNN accelerators. Finally, we evaluate our proposal on E-PUR and TPU. Our experimental results show that compared to the state-of-the-art RNN batching techniques, E-BATCH improves throughput by 1.8x and energy-efficiency by 3.6x in E-PUR and by 2.1x and 1.6x in TPU, respectively.

7.1 RNN Batching

Nowadays, a plethora of machine learning applications using RNNs are evaluated in the cloud. In this domain, GPUs and high-performance hardware accelerators such as TPU [51] and BrainWave [29] are employed. As mentioned earlier, since RNN inference exhibits a limited amount of parallelism, the resource utilization for these accelerators is low. Specifically, it is 18% for TPU and 3.5% for BrainWave for a batch size of one input sequence. GPUs/CPUs using state-of-the-art RNN libraries also exhibit extremely low resource utilization. As an example, the high-performance library cuDNN [11] shows an average utilization of 13.5% on an NVIDIA Titan V GPU for RNN inference. Note that, for LSTM inference, although the amount of computation (matrix-vector multiplications) increases as the number of time-steps in the input sequences increase, the parallelism is limited due to data-dependencies.

Servers handling multiple requests (i.e., cloud services) from a large number of edge devices, such as smartphones, employ batching to increase parallelism and throughput. During inference,

batching merges several requests and feeds them to the system simultaneously, so that all their computations are done in parallel. Hence, all the requests in a batch share the high cost of accessing the model parameters. Note that batching works best when batched requests are identical in length, i.e., the number of time-steps are the same. However, this is particularly difficult in RNNs since their input sequences usually have different number of time-steps. For instance, Deepspeech2 [7] has input sequences with a number of time-steps that ranges from 60 to 1700 (Librispeech test set).

State-of-the-art deep learning systems [2, 77] handle this issue by padding the batched sequences such that their number of time-steps are identical to the number of time-steps of the longest sequence. The main drawback of this approach is that the latency of all the batched sequences increases since their evaluation cannot be completed until the longest sequence has been evaluated. Besides, energy is wasted performing computations on the extra added time-steps. Our experiments on E-PUR show that 30.2% of the energy consumption is due to padding, whereas the latency overhead is 28.5% on average for a set of RNNs.

Moreover, we observe that when evaluating Deep RNN models on accelerators, weight reuse is severely affected by the number of requests in a batch and the number of time-steps in the batched sequences. The reason is that when evaluating a new layer of an RNN model, the model's weights are first brought to on-chip memory, hence evicting the weights of the previous layer. Henceforth, creating batches with short sequences (i.e., low number of time-steps) incurs in a large amount of weight swapping, and, as a consequence, energy consumption increases. This issue is evident in batching strategies such as Cellular Batching [33], where batches with short sequences are created, resulting in large increase in energy consumption due to the frequent movement of weights from off-chip to on-chip memory. For instance, cellular batching on top of E-PUR consumes, on average, 4.5x more energy per request than sequence padding for DeepSpeech [7].

Motivated by the inefficiencies of current batching schemes, we propose E-BATCH: an RNN batching scheme that improves energy efficiency by avoiding padding and by increasing the temporal and spatial locality of the weights. In the next sections, we describe the state-of-the-art strategies used during RNN batching. Also, we present the primary sources of energy inefficiencies for RNN batching on hardware accelerators and E-BATCH.

7.1.1 Batching Strategies

As mentioned in Section 2.3.3, Sequence Batching (batching for short) is a well-known technique that is commonly used to increase throughput. A batch is a set of one or more requests (i.e., input sequences), and the number of batched requests (i.e., batch size) is usually limited by the amount of hardware resources available in the system. In this work, we refer to the processing elements employed to evaluate one or more input sequences as *processing lane*.

Traditionally, arriving requests are grouped into batches of size N, and batches are evaluated sequentially. Commonly, once a batch of requests is sent to the hardware for evaluation, each one of the batched requests is assigned to a processing lane. Also, the evaluation of any of those requests is not completed until all of them are computed. Henceforth, batching tends to work best when the batched requests have the same length (i.e., an identical number of time-steps among input sequences) since all the processing lanes are fully utilized, and none of the batched requests

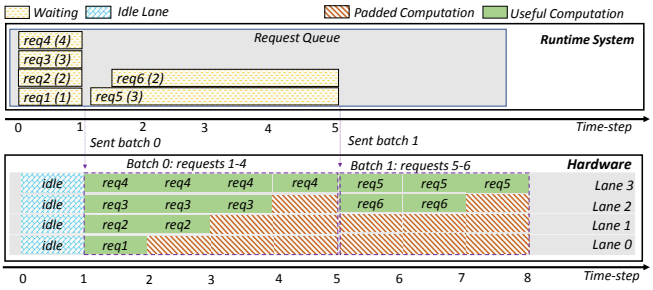


Figure 7.1: Sequence Padding. Requests are shown in the request queue from their time of arrival until they are dispatched to the hardware for evaluation. The number inside the parenthesis next to each queued request is the number of time-steps for that request. The batch size is 4.

will have to wait for others to complete. However, this is an issue in RNNs since the number of time-steps on each input sequence is typically different. Henceforth, to mitigate this problem, the following strategies are commonly employed.

Sequence Padding

Sequence Padding is used in systems such as TensorFlow [2] and PyTorch [77] in order to handle sequences with different number of time-steps when batching is performed for RNNs. In this case, the sequence with the largest amount of time-steps (i.e. m) in a batch is found and, for each of the other sequences in the batch, their number of time-steps is increased to match the maximum length m. These extra added time-steps are filled with zeros, and the larger the difference among batched input sequences, the larger the amount of useless computations. As an example, let us consider the batches created in Figure 7.1, where *requests 1-6* have 1, 2, 3, 4, 3 and 2 time-steps, respectively. Hence, in order to create a batch of size 4, *requests 1-3* are padded. Note that *request 1* completes its execution long before *request 4* (i.e. the longest request in the batch), but it is not returned to the user until *request 4* is finished. Furthermore, although *request 5* is available when *request 1* is already finished, its computation cannot start until the whole batch is computed.

Sequence Bucketing

Sequence Bucketing (bucketing) [55] is an optimization technique that is implemented on top of sequence padding in systems such as MXNet [18] and TensorFlow. Its target is to reduce padding, hence decreasing the amount of wasted computations. In order to accomplish this, different sequences are clustered together into a logical group, a.k.a. *bucket*, based on a given heuristic. One commonly used heuristic, shown in Figure 7.2, is to assign sequences of similar length to a given *bucket*. Similarity is defined as the maximum difference in time-steps among all the sequences in any given *bucket* and is constrained to be below a given threshold (i.e., the *bucket width*). Then, when a batch is created, only sequences from the same *bucket* can be batched together. Note that some sequences in a given batch may still require some padding. The maximum amount of time-steps padded is the *bucket width*. For instance, let us consider the example in Figure 7.2, assuming that *requests 1-6* have 1, 2, 4, 5, 3, and 2 time-steps respectively, and the *bucket width* is one. *Requests*

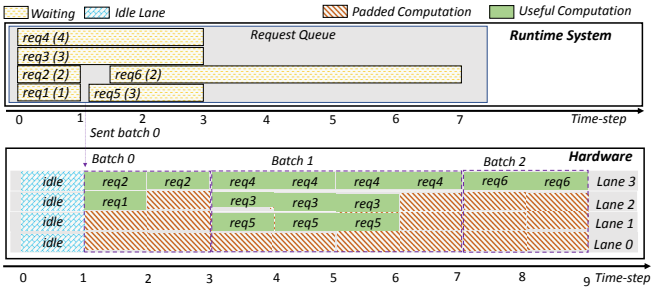


Figure 7.2: Sequence Bucketing. For this example, the maximum difference in time-steps for requests that are batched together is 1.

1, 2, and 6 are assigned to a *bucket* whereas *request 3-5* are assigned to another *bucket*. Then, if batches are created using a batch size of 4, *requests 1-2* will be batched together. Similarly *requests 3-5* will go into the same batch. Note that, although *request 6* is available when *requests 3-5* are batched, it is not included since it belongs to a different *bucket*. Analogous to sequence padding, batches are evaluated sequentially, and new requests are not allowed to join a given batch while it is being evaluated in the hardware.

Cellular Batching

Cellular Batching is a recently proposed technique for RNN batching that focuses on batching requests at the granularity of cells (i.e., several time-steps) instead of whole sequences [33]. Unlike sequence padding and bucketing, in cellular batching new requests are allowed to join a batch whose execution is ongoing. Also, once a request is evaluated, it can be returned to the user immediately. Figure 7.3 shows a simple example of how requests are scheduled in cellular batching. For this example, we assume a cell represents one time-step, a batch size of 4 and an LSTM model of one layer. In this example, the first time-steps of *requests 1-4* are batched together and sent to the hardware for evaluation. Once the batch is evaluated, *request 1* is completed and returned to the user. After this, a new batch is created using the second time-step from *requests 2-4* and time-step 1 from *request 5*. Once this batch is evaluated, *request 2* is completed and sent to the user. Then, the process of batching and evaluation continues for the remaining time-steps of *requests 3-6* until all of them are evaluated. Since batches are created using a fine granularity, new requests can start execution as soon as the processing hardware becomes available, and completed requests are sent to the user immediately. Hence, the time a request waits in the queue is reduced. Thus, improving the average latency and throughput of the system.

7.2 Source of batching inefficiencies in RNNs

Both Bucketing and Cellular batching provide an improvement over sequence padding by reducing the amount of wasted computations. Cellular batching also improves latency by batching requests using a finer granularity. However, these solutions do not take into account energy consumption and the spatial and temporal locality of the weights for large RNN models (i.e., more

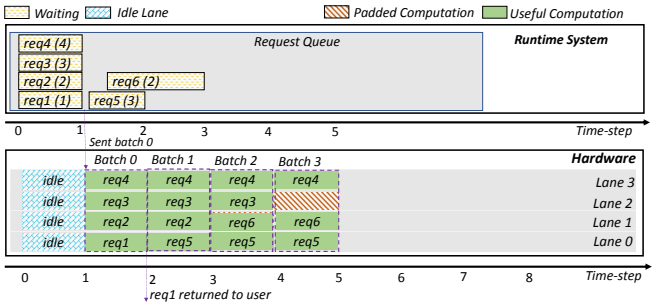


Figure 7.3: Cellular Batching.

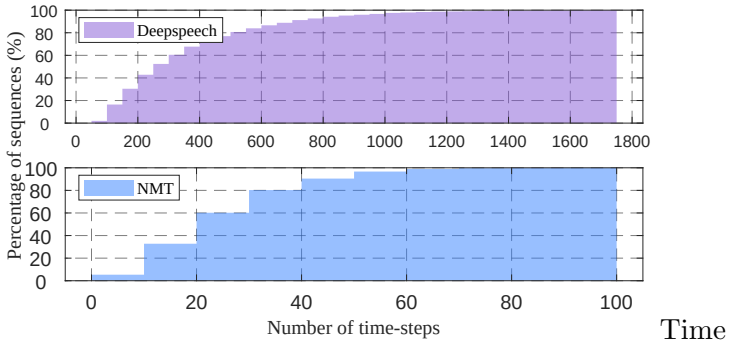


Figure 7.4: Time-steps distribution for Deepspeech [7] and NMT. The number of time-steps ranges from 10 to a few hundred.

than one layer), which is typically exploited in RNN accelerators. In this section, we identify the sources of inefficiencies in RNN batching and present a detailed analysis.

7.2.1 Number of Times-steps Variability

Figure 7.4 shows the cummulative distribution for the number of time-steps in the input sequences of two popular RNN models. As it can be seen, for both models, there is a wide range of sizes in the length of their input sequences. This variability severely affects batching systems that employ strategies such as sequence padding or bucketing since many useless computations are introduced. For instance, we have seen in our experiments that for sequence padding, nearly 40% of the computations evaluated by the hardware are useless. On the contrary, when bucketing is applied, the number of wasteful computations evaluated decreases to nearly 5% since the number of time-steps among sequences that are batched together is quite similar. Regarding cellular batching, less than one percent of the computations executed are useless since batches are created and evaluated at a fine granularity (i.e., five time-steps).

7.2.2 Low Hardware Resource Utilization

There are two leading causes of low hardware utilization of RNNs systems. One is due to padded sequences whereas the other is due to a limited amount of requests.

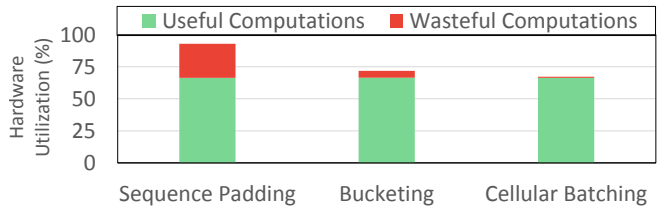


Figure 7.5: Percentage of hardware utilization for useful and wasteful computations when evaluating Deepspeech [7] on an accelerator.

Figure 7.5 shows a breakdown for the percentage of hardware utilization for useful and wasteful computations on a system under a moderate workload (i.e., 1000 requests per second). As it can be seen, when employing sequence padding around 26% of the hardware utilization is due to wasteful computations. On the contrary, when bucketing is employed, only 5.2% of the hardware is utilized for wasteful calculations.

Some wasteful computations are performed by lanes that complete the evaluation of requests assigned to them early. For these cases, we could reduce the number of wasteful computations by sending available requests to those lanes, since there are no data dependencies among requests. Note that typically when a batch is being evaluated in the hardware, the same model parameters (i.e., weights) are used by each processing lane to increase weight reuse. Therefore, time-steps from an available request can only be sent to a lane when its weights are the same as the weights being used to compute the current set of batched requests. We provide more details about this issue in Section 7.3.2.

On the other hand, when the number of requests waiting to be processed is not enough to fill the available lanes, batches of small sizes are created. For this reason, some processing lanes will perform padded computations. However and, more importantly, weights are swapped more frequently (i.e., for each new batch and layer evaluated), henceforth decreasing weight reuse and dramatically increasing energy consumption. For instance, for the model evaluated in Figure 7.5 when the workload is low (i.e., the number of requests is smaller than the batch size), waiting for more requests to arrive to create larger batches decreases energy consumption per request processed by 2.3x on average.

7.2.3 Poor Weight Locality

One of the primary sources of energy consumption for RNN inference is the memory activity to fetch the weights. Thus hardware accelerators for RNN include local on-chip memories to increase weight reuse. Not surprisingly, even when batching RNN sequences weight fetching is still the dominant factor in energy consumption.

For deep RNNs, on each batch evaluation the weights must be fetched for each layer. Note that while evaluating an RNN layer, due to data dependencies, all the time-steps of a group of sequences in a batch are usually computed before proceeding with deeper layers. Hence, depending on the number of time-steps of the batched sequences, the overhead of accessing the weights will be more or less severe. It is particularly problematic when evaluating cellular batching on accelerators

because input sequences are split into multiple batches. Moreover, batches where the sequences have a small number of time-steps (e.g., 5) are created under cellular batching. Therefore, for RNN models with more than one layer, weights are fetched multiple times for each sequence.

One approach to increase weight reuse is to use large batch sizes so that more requests are batched together. However, increasing the batch size will also increment the number of lanes required to evaluate them. Another approach is to create batches in which several requests are concatenated before being sent to a processing lane, where they are evaluated sequentially. The main drawback of this approach is that it increases the number of time-steps computed sequentially per processing lane for a given batch, which increases the average latency. However, significant improvements in energy efficiency can be achieved (as shown later in Section 7.4).

To improve energy efficiency in RNN batching, we develop E-Batch, a novel batching scheme that allows RNN sequences to join a batch dynamically while the batch is being executed. Our scheme also strives to create batches with a low amount of padding by employing a multi-way greedy partition algorithm. Furthermore, to increase weight reuse, E-Batch tries to evaluate a large number of time-steps in each lane. Finally, to improve resource utilization, E-Batch implements a timeout so that more requests are batched together during intervals when the number of requests arriving at the system is low. We detail E-BATCH in the rest of this chapter.

7.3 RNN Batching on Accelerators

In this section, we present the details of E-Batch. We implemented our proposal on top of E-PUR and TPU. First, we provide the extensions added to E-PUR to support batching. Note that batching is already supported in TPU. Second, E-Batch is described. Finally, we discuss the runtime and hardware support required by our proposal.

7.3.1 Supporting Batching on E-PUR

The base architecture of E-PUR does not support batching. Two main modifications are done to support the evaluation of multiple input sequences in a batch. First, E-PUR includes an on-chip memory to store intermediate results, which saves accesses to main memory. Regarding the intermediate results, most of them are generated after computing the output vector (h_t) since it is used as input to the next time-step of execution and as input to the next layer. However, including an on-chip memory for intermediate results during batching is unfeasible since the memory requirements will be huge (hundreds of megabytes). Note that the output generated on each time-step must be kept in memory since it is consumed by the next layer and, hence, intermediate results quickly become large when increasing batch size. Hence, a vast amount of intermediate results are generated for large batch sizes (e.g. $time\_steps * batch\_size * model\_neurons$). Consequently, intermediate results among RNN layers are stored in the main memory. Despite this, RNN batching improves weight locality largely since it reduces main memory accesses for fetching weights. This trade-off is highly lucrative since the memory footprint for the weights is typically much larger than the size of the intermediate results. For instance, on E-PUR, when sequence padding is employed

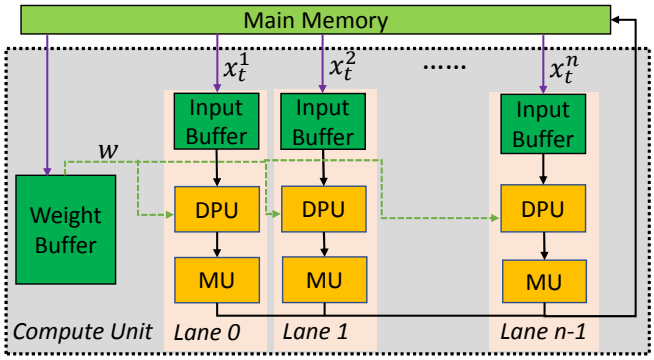


Figure 7.6: Architecture of a Compute Unit (CU) in E-PUR. Several processing lanes are included to support batching.

with a batch size of 64, energy consumption and performance are improved by 3.15x and 36x on average, respectively.

The second extension to the baseline architecture is the addition of extra DPUs and MUs on each CU, as shown in Figure 7.6. On each CU, we include N DPUs and MUs such that we have the necessary hardware required to evaluate N input sequences in parallel (for a batch of size N). Hence, in E-PUR a processing lane consists of a DPU and an MU. Regarding the on-chip memories, each processing lane includes its private input buffer, whereas the weight buffer is shared among all the processing lanes.

After these modifications, the evaluation of a batch is performed as follows. First, once a batch of sequences is created by the runtime (discussed in Section 7.3.4), it is sent to E-PUR. In the accelerator, each input sequence in the batch is distributed to a lane. Then, on each processing lane, the evaluation of a sequence is done, as explained in Section 4.1. Note that since all the processing lanes share the weight buffer, weights are broadcast to each of them where they are multiplied by their corresponding input sequence. Finally, once the output h_t of an input sequence is computed for each processing lane, the result is stored in the main memory.

7.3.2 E-Batch

In section 7.2 we describe the primary sources of inefficiencies in RNN batching. To solve these issues, we propose E-Batch, a novel RNN batching scheme designed to improve both latency and energy consumption. In E-Batch, the creation of a batch is managed on the CPU by a runtime system, whereas the RNN computations are done in an RNN accelerator (E-PUR, TPU, ...). In contrast to other solutions, in our scheme batches are created taking into account all the available requests. Since our main target is energy efficiency, E-Batch increases weight reuse by using all the available requests. When the number of requests being batched is larger than the number of available lanes, multiple requests are assigned to a given lane and are executed sequentially. In this case, the number of time-steps for a given lane is defined as the sum of the time-steps of all the requests assigned to that lane.

Since the number of time-steps for each request tends to be different, the number of time-steps

on each lane also differs. To decrease the amount of padded computations (see Section 7.2.1), our scheme tries to minimize the difference in time-steps between each lane by distributing the available requests among lanes so that their number of time-steps are as similar as possible. Note that this problem is an instance of the Multi-Way Number partitioning problem, which is known to be NP-Complete [59]. We employ a greedy heuristic to assign each request to a given lane. For example, given the requests with the following number of time-steps (2,4,6,8,7) and a system with two lanes, we first sort all the available requests in descending order by their number of time-steps: (8,7,6,4,2). Next, we process the requests sequentially in descending order, assigning each request to the lane with the minimum amount of time-steps (e.g., 8 to the first lane, 7 to second, 6 to the second, and so on). In the example above, the first lane receives requests with time-steps (8,4,2), and the second lane gets requests with time-steps (7,6).

Greedy partitioning does not always yield an optimal solution [59]. As a result, some sequences may need to be padded. In this case, we use hardware assistance to either power gate the lanes that will perform the evaluations of the padded sequences or assign new requests to these lanes. To this end, when a lane finishes the evaluation of all the time-steps assigned to it, E-PUR queries the runtime for a new request and, assuming that a request is available, the runtime sends it to E-PUR where its evaluation starts immediately to avoid padding. On the contrary, if there is no request available, the lane is power gated. Due to data dependencies, time-steps must be evaluated in order and cannot be distributed to multiple lanes.

To reduce the impact of padding and improve throughput, E-Batch allows new requests to join a batch, but only while the first RNN layer is being processed. If a new request arrives at the server while the accelerator is processing the first (i.e., input) layer of an RNN, E-batch's runtime system tries to add the request to the current batch to reduce padded time-steps. Once the computation of the first layer for a given batch is finished, the batch is locked, and it cannot be modified for the subsequent RNN layers. The rationale behind this decision is to guarantee that all the requests in a batch belong to the same RNN layer and are processed at the same time to maximize weight reuse, which results in significant energy savings, as shown in Section 7.4.

In E-Batch, the number of time-steps processed by a given lane is limited to a threshold (i.e., N). By using a threshold, we can trade latency for energy consumption. Batching with a small number of time-steps per lane reduces latency, but incurs in a large amount of weight swaps, which decreases weight locality. On the contrary, batching with a large number of time-steps per lane increases weight reuse, which significantly reduces energy consumption at the expense of an increase in latency.

When a batch executes the first layer of the RNN, E-Batch proceeds to evaluate the next layer after N time-steps have been evaluated per lane. If the amount of time-steps in a lane is larger than N, the remaining time-steps for the requests assigned to that lane are evaluated in a future batch. In other words, the evaluation of some requests is split among different batches to allow requests to make progress through the next layers in the RNN and reduce latency.

When a new batch needs to be started, and the number of requests available is smaller than the batch size, the runtime waits for T milli-seconds to increase hardware utilization and weight reuse (i.e., more requests are batched together) at the expense of some penalty in latency. Furthermore, if a batch is being executed and a new request arrives while the first layer of the RNN is being

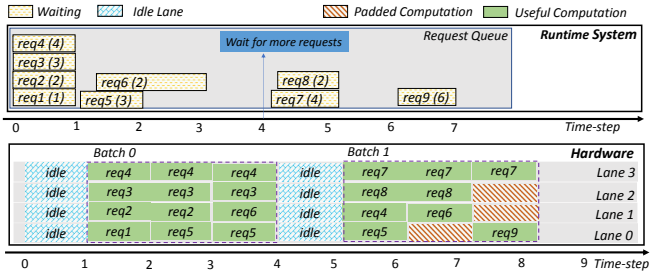


Figure 7.7: At time-steps 2 and 3, *lane 0* and *1* become idle and the evaluation of *requests 5 and 6* starts. The maximum amount of time-steps per lane (N) is set to 3.

processed, the runtime checks for any idle lane, and if there is any, the new request is sent to the accelerator where it is assigned to that lane. Note that, to evaluate layers after the first one is computed, the result of the previous layer is needed. Therefore, even though some lanes could be idle, if a new request arrives when a batch is being evaluated for any subsequent layer, it cannot be assigned to any of those idle lanes. Also, only requests for the same RNN model are batched together.

Figure 7.7 illustrates the execution flow of E-Batch. Similar to the previous examples, a batch size of 4 is used and an LSTM model with one layer. Also, a value of 3 time-steps is used for N, whereas T is set to two time-steps. At the beginning, *requests 1-4* are in the waiting queue. Since the number of requests in the queue is four, a new batch (*batch 0*) is created by the runtime, assigning each request in the batch to a given lane using Greedy partitioning. Afterwards, *batch 0* is sent to the accelerator in time-step 1. At time-step 2, the execution of *request 1* is completed. Hence, the accelerator sends a signal to the runtime to indicate that *lane 0* is idle. Thus, *request 5* is assigned to *lane 0* since it is the oldest request in the waiting queue. Similarly, *request 6* is assigned to *lane 1* after *request 2* is finished. At time-step 4, the execution of *batch 0* is completed since N is 3. At this point, *requests 1-2* are completed, and the RNN output is sent back to the client. However, for *requests 4-6* only 3, 2, and 1 time-steps have been evaluated. Therefore, the remaining time-steps will be computed in a future batch. After the evaluation of *batch 0*, only *requests 4-6* are in the queue. Hence, since the number of requests in the queue is smaller than the number of lanes, the runtime waits for T time-steps or until there are enough requests to fill all the lanes.

As seen in Figure 7.7, at time-step 5 *request 4-8* are in the queue, hence *batch 1* is created by the runtime. In this case, since the number of requests in the queue is greater than the number of lanes, *requests 4 and 6* are assigned to lane 1. Finally, *batch 1* is sent to the accelerator, where its evaluation is performed in a similar manner to *batch 0*.

Figure 7.8 shows the behavior of our scheme for RNN models with more than one layer. Similar to the previous example, for time-step 1, *requests 1-4* are batched and sent to the accelerator. Furthermore, at time-step 2, *request 1* is completed and *request 5* is assigned to *lane 0*. Since, N is 3, after three time-steps have been evaluated, we proceed with the evaluation of the next layer. Note that *requests 6-7* are available at time-step 4 and that lane 1 is idle after time-step 6. However, as mentioned before, new requests cannot join a batch after the evaluation of the first layer is completed. Hence, *requests 6-7* wait until the evaluation of *batch 0* finishes for the second

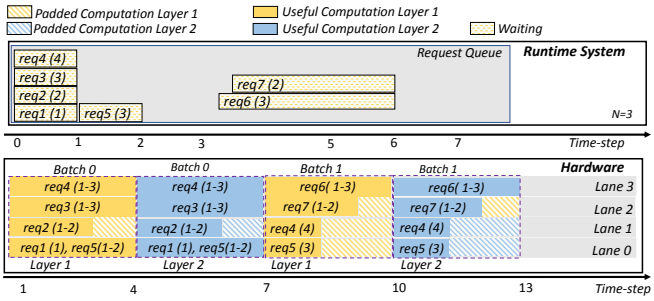


Figure 7.8: Evaluation of an RNN with two LSTM layers using E-Batch. Batch 0 is evaluated for the first and second layer, before a new batch is created. *req 5* arrives before finishing the evaluation of the first layer and, hence, it joins Batch 0 immediately. On the contrary, *req 6-7* arrive after the evaluation of the first layer, so its evaluation is deferred until a new batch is created.

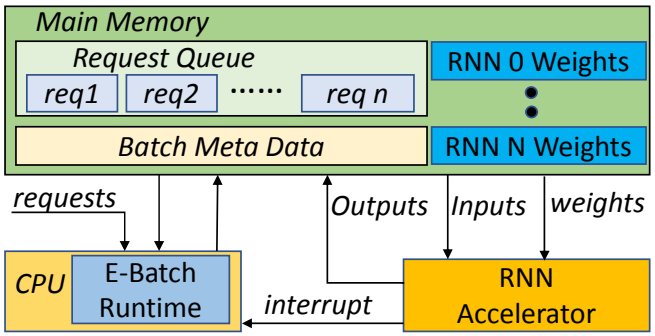


Figure 7.9: Overview of E-Batch System Architecture.

layer. After *batch 0* is completed, the output results for *requests 1-2* are sent to the user, whereas a new batch is created using the remaining time-steps of *request 4-5* and the new requests *request 6-7*.

The overall architecture of E-Batch is shown in Figure 7.9. It is composed of a runtime system and an RNN accelerator. The runtime is in charge of creating and managing batches of requests, whereas evaluations are performed in the accelerator. In the next subsections, we describe in more detail these components.

7.3.3 Hardware support for E-Batch

Supporting E-Batch on E-PUR

The following modifications are done in E-PUR to support E-BACTH. First, we include an interrupt, which is used to signal the runtime when a lane becomes idle. Furthermore, we include a small buffer (i.e., *the request buffer*) to keep track of the lane where each request is being evaluated. Also, in *the request buffer*, we store the number of time-steps that have to be processed for each request. Finally, we include a register to store the parameter N (i.e., maximum number of time-steps). Once a batch is received, the evaluation of the first layer works as follows. For each request

in the batch, an entry in the *request buffer* is created and initialized with its meta-information. Then, when a time-step is evaluated, the number of time-steps processed for the active requests are updated in the *request buffer*. In the case that all the time-steps for a given request are completed, we proceed with the next request in the lane. However, if there are no more requests in the lane, a signal is sent to the runtime to indicate that a lane is idle so that a new request can join the current batch being executed. Then, after N time-steps have been evaluated, we continue with the computation of the next layers. Finally, once the evaluation of a batch is completed, the number of time-steps evaluated for each request is sent back to the runtime.

Supporting E-Batch on a TPU-Like Architecture

TPU is a state-of-the-art accelerator for neural networks [51]. It is composed of a systolic array of processing elements (PEs) and on-chip memory for weights and activations. In order to evaluate LSTM models on TPU like architectures, an output stationary dataflow is employed [81, 87]. In this regard, for an RNN model with N neurons, neuron n_k will be mapped to all the PEs in column k of the array. Regarding the weights, they are mapped to the columns of the systolic array, whereas the input sequences are mapped to rows of the array (one input sequence per row). Thus, to evaluate n_k, its weights are streamed down through column k, whereas the elements of each input sequence are streamed through the rows. Note that the maximum batch size supported is equal to the number of rows in the array and that each of them will correspond to a *processing lane*. To support E-Batch in a TPU-like architecture, as done in E-PUR, we add an interrupt to signal the end of a sequence to the runtime and a *request buffer* as described in 7.3.3.

7.3.4 Runtime support for E-Batch

The runtime is in charge of the management and creation of batches. It includes a queue where new requests arriving at the system are stored. Furthermore, for each request, the runtime tracks the number of time-steps that have been evaluated. Also, it knows whether the accelerator is processing or idle.

When the accelerator is idle, the system will create a new batch using all the available requests by employing the Greedy partitioning algorithm described in Section 7.3.2. Then, when a batch is created, for each batched request, the lane assigned to it and the number of time-steps that need to be evaluated are sent to the accelerator. Once the evaluation of a batch is completed, the number of time-steps evaluated are updated for all its requests. Moreover, requests that are completed (i.e., all their time-steps have been executed) are returned to the user. On the other hand, new requests and requests with time-steps pending for evaluation are batched together. If the available requests are not sufficient to create a complete batch, the runtime will wait until the batch is completed or T milli-seconds have passed.

Table 7.1: Hardware configuration for E-PUR and TPU.

E-PUR	
Parameter	**Value**
Technology	28 nm
Frequency	500 MHz
Weight Buffer	2 MiB per CU
Input Buffer	128 KiB per CU
DPU Width	64 operations
TPU-like accelerator	
Frequency	700 MHz
SRAM Buffer	24 MiB
Systolic Array PEs	128x128

7.4 Experimental Results

In this section, we describe the experimental evaluation of E-BATCH. We employ the methodology presented in Chapter 3 to model E-PUR and a TPU-like architecture. The configuration parameters for E-PUR and TPU are shown in Table 7.1. As benchmarks, we use the machine translation [110] and the speech recognition [7] models shown in Table 3.4.

Regarding the runtime, we implemented a system that employs the timing estimation and status of the accelerator' simulator (E-PUR or TPU) to batch new requests. For our experiments, we simulate several hours of the system execution. To this end, the train/test datasets are not sufficient since they would be processed in a short time span. Aiming to generate more requests to be processed by our system, we analyzed the distribution of the number of time-steps for each input sequence in the train and test set of the RNN models in the benchmarks. Based on this analysis, we generate requests whose number of time-steps follows the distribution observed in the original datasets. Note that the execution and energy consumption of a given request only depends on the RNN parameters and the number of time-steps. To reproduce a real environment, we simulate the arrival time of each user's request. To this end, we use a Poisson distribution. To increase or decrease the number of requests per second arriving to the system (i.e., system load), we change the average inter-arrival time between requests. Note that, for each of our experiments, the average request arrival rate is kept constant. For the rest of this section, we detail the evaluation of our proposal on E-PUR and a TPU-like architecture. Regarding the batch size, we tested several batch sizes and found that the results are similar. Thus, we chose 64 as it delivers the best trade-off between performance and area for E-PUR, whereas for a TPU-like architecture we use 128. First, we discuss the evaluation of bucketing and cellular batching on E-PUR. Second, we discuss performance and energy consumption of E-Batch for E-PUR. Finally, we discuss the results of E-Batch for a TPU-like architecture.

7.4.1 Sequence bucketing and cellular batching on E-PUR

To evaluate the performance of bucketing and cellular batching, we implement them on E-PUR and use an LSTM model with two layers and 800 neurons, whereas the input sequences were drawn from the Deepspeech benchmark. Our experiments show that sequence padding is on average 7x faster than bucketing and delivers 1.6x higher requests per joule. As discussed in Section 7.2.2, bucketing with a small *bucket width* tends to create batches with a low number of requests and, thus, weights are swapped more frequently resulting in larger energy consumption than sequence padding, which manages to create batches with a larger number of requests. Moreover, with bucketing, requests have to wait longer before they are sent to the accelerator, thus increasing their latency. Note that, when the number of requests per second arriving to the system is low bucketing and padding achieve similar results since they behave similarly.

Regarding cellular batching, it achieves, on average, a 1.2x speedup over padding, but energy efficiency is much worse: 4.5x reduction in requests per joule. Furthermore, the maximum throughput achieved by cellular batching is similar to sequence padding because of the large amount of weights that are evicted from on-chip memory each time a new batch of requests is evaluated. Weight eviction occurs because batches composed of requests that belong to different RNN layers do not share their parameters. Cellular batching works better on GPUs since it was not designed to consider the high degree of on-chip locality commonly found in RNN accelerators.

Both bucketing and cellular batching achieve a throughput similar to padding. However, padding is more energy efficient. Therefore, for comparison purposes, we chose an accelerator (E-PUR or TPU) employing sequence padding as the baseline configuration.

7.4.2 E-Batch on E-PUR

Figures 7.10 and 7.11 show the average latency per request for Machine Translation and Speech Recognition RNNs, respectively, on an E-PUR-like accelerator. Both figures include the baseline and our E-batch scheme under different loads and using 64 lanes (maximum batch size). For E-Batch, we use different values of the threshold N (i.e., maximum number of time-steps in a lane) to evaluate the trade-off between energy and latency. In these plots and others, $N = 0$ refers to the E-Batch configuration where the maximum number of time-steps in a lane is set to the number of time-steps of the longest sequence in the batch when it is created. The Speech Recognition network has a larger average latency since its input sequences are typically larger than the input sequences of the Machine Translation model. Therefore they result in more considerable processing and memory transfer times.

As it can be seen for both networks, when N is zero the latency of using sequence padding and E-Batch are similar because, in this case, the longest sequence in a given batch is the same for both schemes. However, since E-Batch can add new requests to a batch while it is being processed, the queuing time decreases, henceforth reducing the average latency. Note that for low loads (i.e., 100 requests per second in Figure 7.10), the average latency is sightly higher for E-Batch. It is because E-batch waits for some time to increase the number of requests that are batched together, thus improving weight locality and energy efficiency.

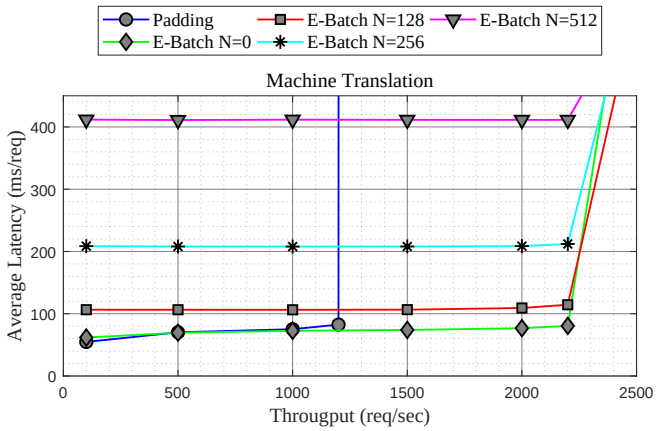


Figure 7.10: Average Latency vs throughput for Machine Translation [110] using a batch size of 64 on E-PUR.

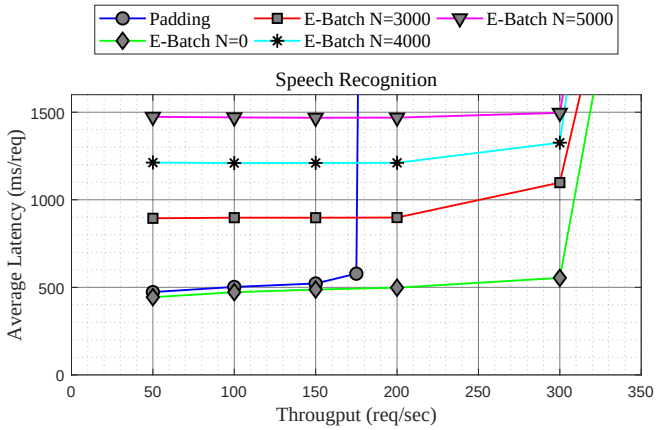


Figure 7.11: Average Latency vs throughput for Speech Recognition [7] using a batch size of 64 on E-PUR.

Regarding the maximum throughput, as it can be seen in Figures 7.10 and 7.11 E-Batch achieves 1.83x and 1.77x improvement over padding for Machine Translation and Speech Recognition, respectively. This improvement in throughput comes from allowing new requests to start execution while the first layer of an RNN is being evaluated. In addition, because of the variability in the number of time-steps, when the evaluation of small requests is completed, a new request from the waiting queue can join the current batch being executed. Therefore, hardware utilization is increased, improving system throughput. The maximum throughput obtained by E-Batch for different values of N in Figures 7.10 and 7.11 are similar, slightly increasing when N becomes very large.

Figures 7.12 and 7.13 show the average number of requests per joule for Machine Translation and Speech Recognition, respectively. As it can bee seen, when N is zero sequence padding tends to have a slightly better energy efficiency than E-Batch for low loads. It occurs because when the number of requests in a batch is small, as new requests arrive at the system, they are added to the batch being processed. However, since N is 0 once the time-steps for the longest and oldest

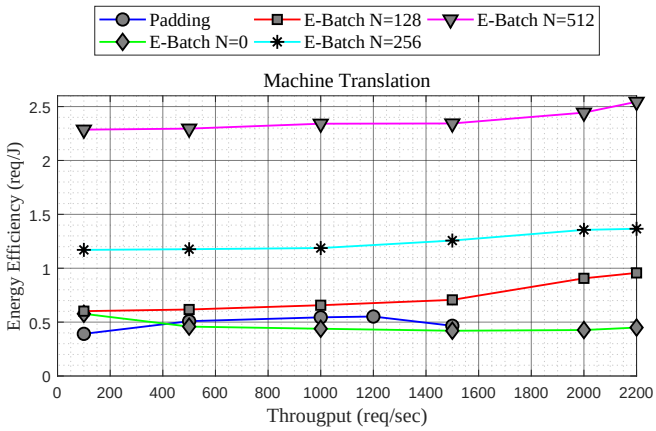


Figure 7.12: Average number of Requests per Joule vs Throughput for Machine Translation [110] using a batch size of 64 on E-PUR.

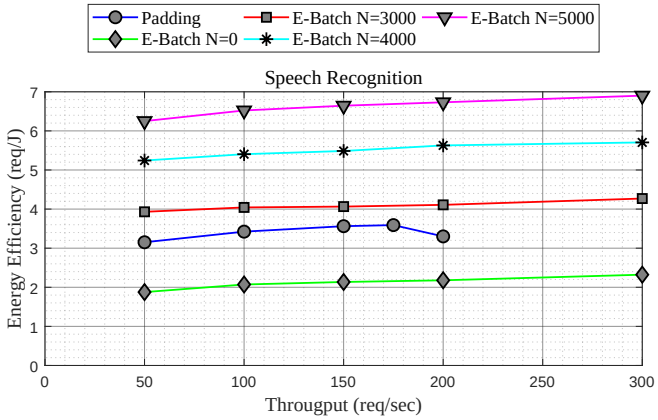


Figure 7.13: Average number of Requests per Joule vs Throughput for Speech Recognition [110] using a batch size of 64 on E-PUR.

request in the batch are computed, the system proceeds to evaluate deeper layers. As a result, the computations of some requests are divided among several batches, and thus they require more memory accesses to fetch the weights. On the contrary, as the load increases, batches with a larger amount of time-steps are created, and thus weight reuse increases.

As shown in Figures 7.12 and 7.13, increasing the value of N will dramatically increase the energy efficiency of E-Batch for both networks. The reason is that since the system waits until N time-steps are evaluated in the first layer, a large amount of time-steps are batched together and, as a result, weight reuse is increased.

7.4.3 E-Batch on a TPU-like Accelerator

Figure 7.14 shows the average latency per request for sequence padding and E-Batch running on top of a TPU-like accelerator. The number of lanes used is 128 since this is the number of

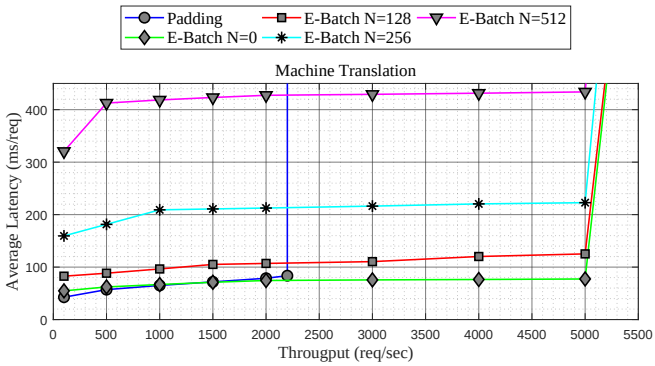


Figure 7.14: Average Latency vs Throughput for Machine Translation [110] using a batch size of 128 for TPU.

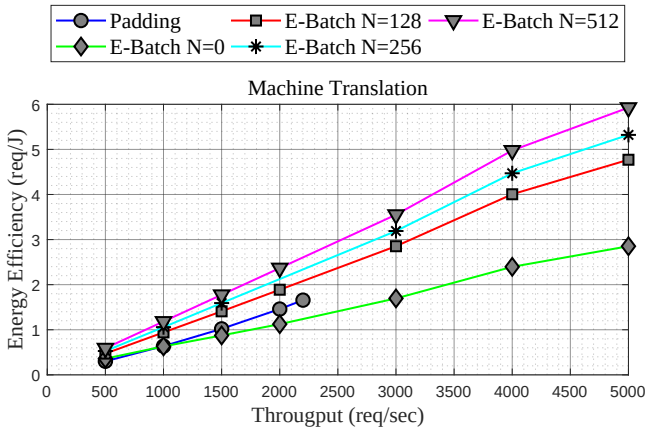


Figure 7.15: Average number of Requests per Joule vs Throughput for Machine Translation [110] using a batch size of 128 for TPU.

rows of the systolic array in the TPUv1. As it can be seen, when N is zero the average latency of sequence padding and E-Batch are similar since, in this case, the largest number of time-steps for a given batch is the same in both schemes. However, since E-Batch allows new requests to join a batch while it is being computed for the first layer, the maximum throughput of E-Batch is 2.1x higher.

Figure 7.15 shows the number of requests per joule for the machine translation network evaluated on a TPU-like architecture. Like E-PUR, when the load is small, and N is zero, a large number of batches with a small number of time-steps are created, and as a result, the number of memory accesses increases. However, as the number of requests per second increases, a more substantial amount of requests are batched together, thus increasing weight reuse. Similarly, as N increases, a more considerable amount of time-steps are batched together, which also increases weight reuse. For instance, when N is 128, and the load is 2000 req/sec, E-batch achieves an energy efficiency improvement of 1.3x compared to the baseline, whereas when N is 256 or 512 energy efficiency is improved by 1.46x and 1.6x, respectively. However, note that when improving energy efficiency, the average latency is also increased.

7.5 Related Work

Several hardware accelerators [51, 29, 112, 41] and software libraries [11, 114, 49] tailored to improve energy efficiency and performance of RNNs have been proposed recently. In addition to hardware acceleration, several techniques such as pruning [81, 43], model compression [42], and computations reuse [82, 94] have been employed. Our optimizations are orthogonal to those techniques.

Regarding software libraries such as as [49, 11, 78], they are mainly tailored to GPUs and CPUs [114], whereas our proposal targets specialized accelerators. Previous proposals employ sequence padding or bucketing to handle sequences of different sizes. In CNTK [90], wasted computations are avoided by trying to batch small sequences when they can fit in the padded space. However, this is not always possible since short sequences may not be available. Conversely, in E-Batch, a sequence can be split among different batches so that only the amount of time-steps required to fill up the padded space is used.

Batching RNN sequences is supported in hardware accelerators such as BrainWave and LSTM-Sharp. BrainWave [29] is highly optimized for batches of size one, and inputs are processed sequentially, by computing one single input at a time. In this accelerator, sequence padding is not required. However, employing batch sizes larger than two is unfeasible since inputs are processed sequentially. LSTM-Sharp [112] focuses on increasing resource utilization. It addresses the issue of extra padded computations that occur when performing matrix-vector multiplications, and the number of multipliers per tile are not multiples of the input vector size. This padding is different to padding several sequences to make their sizes homogeneous. Sequence padding is also needed for LSTM-Sharp, to support batching. E-batch is orthogonal to this work and can be implemented on top of this accelerator. TPU [51] supports large batch sizes by means of sequence padding.

7.6 Conclusions

In this chapter, we presented E-Batch, a batching system for recurrent neural networks that increases throughput while improving energy efficiency. E-Batch consists of a runtime and minor extensions to the hardware accelerator. In E-Batch, long sequences are batched together to decrease memory accesses. Furthermore, throughput is increased by allowing requests to join other requests while their execution is ongoing. We evaluated E-Batch on top of E-PUR and TPU, two state-of-art hardware accelerators for RNNs. Our experimental results show that in E-PUR, E-Batch improves throughput by 1.8x and energy efficiency by 3.6x, whereas for TPU throughput is enhanced by 2.1x and energy efficiency by 1.6x.

8

Conclusions and Future Work

In this chapter, the main conclusions of this thesis are presented. We summarize the main contributions and outline some open-research areas for future work.

8.1 Conclusions

The purpose of this thesis is to design energy-efficient architectures tailored to the execution of RNN inference while enhancing performance. In this regard, we develop E-PUR, a highly efficient low power RNN accelerator. Moreover, we implement several techniques that improve the performance and energy efficiency of RNN inference on hardware accelerators.

We first identified the main challenges regarding the evaluation of RNNs in an energy-efficient manner. We observe that most of the energy consumption when performing RNN inference is due to memory accesses to fetch the weights. Moreover, we note that due to data dependencies, the amount of parallelism is limited. Then, we design E-PUR, a custom accelerator tailored to RNN inference. E-PUR is designed to support large RNN models while providing high energy efficiency. E-PUR includes several on-chip memories to mitigate the cost of accessing main memory. Also, it improves performance by overlapping several operations of the RNN evaluation process. Compared to highly optimized software-based solutions running on a modern System-on-Chip, E-PUR increases energy efficiency by 58x on average, while the performance is improved by 6.9x. On average, the power dissipation of E-PUR is 500 mW.

We further improve E-PUR by implementing Maximizing Weight Locality (MWL). MWL performs all the computations involving the forward connections in parallel since they are independent. Then, the recurrent connections are evaluated sequentially. By employing MWL, we reduce the on-chip memory requirement by 2x. Compared to a modern SoC, energy efficiency is improved by

88x, while the power dissipation is 330 mW.

Despite the significant energy savings and performance improvements of E-PUR, we observe that memory accesses are still the primary source of energy consumption. Thus, we propose two novel techniques to further improve the energy efficiency of RNN evaluation. The first technique is based on reusing computations, whereas the latter focuses on changing the bit-width at runtime.

Regarding computations reuse, we implement a fuzzy memoization scheme to cache and reuse the neurons' output. Our technique is based on the observation that the output of a given neuron tends to change slightly between consecutive time-steps. We exploit this observation by proposing a novel memoization scheme that employs a binary neural network as a predictor of reusability. We evaluate this technique in E-PUR. We show that by avoiding an average of 24.2% of E-PUR's calculations and memory accesses, its energy efficiency is improved by 18.5% on average and a speedup of 1.35x is obtained.

RNN inference is commonly performed using low bit-width. In this regard, a worst-case precision is traditionally assumed at the model level or layer level, such that the base accuracy is maintained. We observe that for RNNs, the precision required for computations varies on each time-step. More precisely, empirically we determine that the precision requirements of RNN computations vary according to the value of the cell state. We exploit this observation by proposing a novel technique that changes the precision used during RNN computations at runtime. In this technique, we dynamically switch between the worst-case precision and a lower precision. We show that by tracking the cell state value, on average 57% of the computations can be done in a precision lower than the static worst-case precision while maintaining the base accuracy of the model. We implement this technique on top E-PUR. Our evaluations show that by performing 57% of the computations in a lower precision, energy efficiency is improved by 19.2% on average. Also, performance is enhanced by 1.46x on average.

Finally, we observe that several systems increase energy efficiency, parallelism, and throughput by batching several RNN input sequences so that they are evaluated concurrently. However, batching requires that the input sequences have a similar number of time-steps to be efficient. We analyze the state-of-the-art batching strategies and observe that they incur in a high amount of energy consumption. Hence, as a first step, we identify the primary sources of batching inefficiencies during RNN inference. After this analysis, we determine extra-padded computations, low hardware resource utilization, and evicting weights too frequently as the principal sources of energy consumption during RNN batching. To mitigate these issues, we develop E-BATCH, a novel batching scheme which reduces the amount of padded computations while maintaining the temporal and spatial locality of the weights. We implement it on top of E-PUR and a TPU-like architecture. Our evaluations show that energy efficiency is improved by 3.6x/2.1x on average for E-PUR and TPU, respectively, whereas throughput is enhanced by 1.8x/1.6x, respectively.

8.2 Summary of Contributions

In this thesis, several energy-saving techniques for RNN inference have been proposed. We have observed different properties of RNNs and used them to propose several techniques, from

changing the order of computations to dynamically selecting the bit-width employed during calculations. Moreover, we have introduced a custom architecture and a novel batching scheme. The contributions of this dissertation are summarized as follows.

First, we design E-PUR, a custom accelerator tailored to RNN inference. Our proposal is motivated by the increasing importance of RNNs as a building block of machine learning applications. Our design includes a highly optimized and energy-efficient pipeline tailored to the computations of the matrix-vector multiplications, which are the majority of the calculations performed during RNN inference. It also includes on-chip memory to store one RNN layer, which allows the evaluation of large RNN models while also being low power. In this design, we also propose maximizing weight locality, a novel technique that changes the order of computations to improve weight reuse. The final design provides significant performance and energy efficiency improvements, as described in Chapter 4. This work has been published in the proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques (PACT):

- Franyell Silfa, Gem Dot, Jose-Maria Arnau, Antonio González, "E-PUR: An Energy-efficient Processing Unit for Recurrent Neural Networks," 27th International Conference on Parallel Architectures and Compilation Techniques (**PACT)**, Limassol, Cyprus, 2018.

Next, we propose a novel fuzzy memoization scheme that reuses the memoized output of specific neurons to avoid their computations. In this proposal, we show that a binary neural network (BNN) can be effectively employed as a predictor of reusability. One of our key observations is that the output of a BNN is highly correlated with the full precision counterpart's output. Moreover, we show that between consecutive evaluations the output of a neuron changes by less than 23% on average. We implement and evaluate this proposal on top of E-PUR and show that it provides substantial improvements in terms of energy and performance, as presented in Chapter 5. This work has been published in the proceedings of the 52th international symposium on Microarchitecture (MICRO):

- Franyell Silfa, Gem Dot, Jose-Maria Arnau, Antonio González, "Neuron-Level Fuzzy Memoization in RNNs," 52th IEEE/ACM International Symposium on Microarchitecture (**MICRO)**, Columbus, OH, USA, 2019.

Then, we propose a novel technique that dynamically selects between different precision levels. We analyze the cell state and show that, depending on its current and recent values, we can safely change the bit-width required to perform specific computations. By employing a lower precision, energy efficiency and performance are improved since many memory accesses are avoided, and the execution time of some evaluations is reduced. We implement a scheme that performs 57% of the calculations in a low precision on average, while maintaining accuracy. For this proposal, we employed parallel multipliers that operate with multiple bit-width. Moreover, we evaluate our scheme using serial inner product units since they have a finer bit-width granularity. To track the cell state and dynamically set the precision, we employ a simple yet effective hardware solution with an area overhead of less than 4%. In Chapter 6, we present the energy and performance benefits of our proposal. This work is under revision for publication.

- Franyell Silfa, Jose-Maria Arnau, Antonio González, "Boosting LSTM Performance Through Dynamic Precision Selection,"

As the last contribution, we focus on batching strategies for RNN. We analyze the state-of-the-art approaches for RNN batching to characterize their main drawbacks. Based on this analysis, we propose E-Batch, a batching system for recurrent neural networks that increases throughput while improving energy efficiency. E-BATCH consists of a runtime and small extensions to the base accelerator. One of the primary problems addressed with E-BATCH is the large amount of weight swaps that are performed by state-of-the-art batching approaches. Swapping RNN layers from on-chip memory severely affects energy-efficiency since weights are brought to on-chip memory several times to evaluate the same input sequence. We solve this issue by batching longer sequences together. Moreover, we allow new requests to join a batch while it is being evaluated to increase weight reuse and decrease waiting latency. Our proposal provides significant throughput and energy-efficiency improvements, as detailed in Chapter 7. This work is under revision for publication.

- Franyell Silfa, Jose-Maria Arnau, Antonio González, "E-BATCH: Energy-Efficient and High-Throughput RNN Batching,"

8.3 Future Work

An extension of the work proposed in this thesis would be to extend E-PUR to support sparse RNN models. When evaluating sparse models, the memory access pattern to fetch the weights and inputs is irregular. Thus, some indirection is generally required to evaluate RNN sparse models. We believe that the overhead of the extra hardware required to support sparse models is small, but a future study is necessary to quantify it.

Another interesting future study would be to integrate the memoization and dynamic precision technique in E-PUR and evaluate the overall improvement on performance and energy consumption of the accelerator. Since both of these techniques are orthogonal, we believe that both of them' overall benefits will be similar to their original improvements. However, the model's base accuracy could be degraded since a more considerable amount of error will be added by both techniques. Regarding the hardware units required to support both schemes, we believe that there is a challenge laying out the on-chip memories, since these techniques require a specific access pattern to be close to their theoretical improvements.

An interesting work would be to extend E-PUR for server-class devices. As stated in Chapter 7, this will require to include multiple processing lanes to support large batch sizes. Moreover, to increase the overall performance, the number of hardware multipliers should be increased. We believe that after scaling up the number of multipliers, the Multi-functional will become the bottleneck. Besides, the primary source of energy consumption will shift from on-chip memories to the DPU, due to the increase in the number of computations and the multipliers' static energy. Hence, a more suitable design should be done.

A promising extension of the memoization scheme proposed in Chapter 5 would be to further improve the performance of the BNN predictor. While the reuse potential obtained by the BNN

predictor is significant, it is below the reuse percentage of the oracle. More specifically, the oracle predictor has a reuse percentage of around 50% on average, whereas the BNN predictor achieves 24.2%. We believe that this behavior results from not training the BNN during the training of the full-precision RNN. Note that, in our proposal, we obtained the BNN by extending the already trained full-precision RNN. A possible path would be to include the memoization scheme in the training algorithm, in a similar manner to what is commonly done with pruning algorithms. Then, the full-precision network is retrained with the memoization algorithm embedded in the training until the original accuracy is retained.

Recently, the Transformer [102] architecture is achieving tremendous success when applied to sequence to sequence problems. It employs an attention mechanism to identify global dependencies among input and output elements. In this context, the attention mechanism consists only of matrix-matrix multiplications. Since these operations are very similar to the operations performed by an RNN cell, an interesting extension would be to add the required hardware in E-PUR to support Transformer models. We suspect that the modification to the existing design are minimal and the potential benefits in terms of energy and performance could be significant.

Another interesting research path is to apply the techniques proposed in Chapters 5 and 6 to the Transformer architecture. In the case of Fuzzy Memoization, the first question whether there is a high degree of similarity between computations. Answering this question for Transformers is challenging because the input sequence is not evaluated sequentially. Also, there is not a clear concept of neurons. However, we suspect that there could be similarities among the outputs generated on different layers. Regarding the BNN predictor, we believe it could work since it only depends on the dimensionality of the vectors employed during the computation of the attention mechanism.

Regarding the use of low bit-width and dynamic precision selection, we believe it could be applied, nonetheless it is challenging. Note that Transformers do not have cell state or dependencies among time-steps. Hence, a new heuristic to dynamically change the precision would be required and we believe this may be an interesting extension to our work.

Bibliography

[1] cublas. https://developer.nvidia.com/cublas.

[2] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *CoRR*, abs/1603.04467, 2016.

[3] Umut A. Acar, Guy E. Blelloch, and Robert Harper. Selective memoization. *SIGPLAN Not.*, 38(1):14–25, January 2003.

[4] Daniel Adiwardana, Minh-Thang Luong, David R So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, et al. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020.

[5] Carlos Álvarez, Jesús Corbal, Esther Salamí, and Mateo Valero. On the potential of tolerant region reuse for multimedia applications. ICS '01, pages 218–228, 2001.

[6] Carlos Alvarez, Jesus Corbal, and Mateo Valero. Fuzzy memoization for floating-point multimedia applications. *IEEE Trans. Comput.*, 54(7):922–927, July 2005.

[7] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. Deep speech 2: End-to-end speech recognition in english and mandarin. *CoRR*, abs/1512.02595, 2015.

[8] Alexander G. Anderson and Cory P. Berg. The high-dimensional geometry of binary neural networks. *CoRR*, abs/1705.07199, 2017.

[9] https://www.gsmarena.com/apple_iphone_11-9848.php.

[10] https://en.wikipedia.org/wiki/Siri.

[11] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. Optimizing performance of recurrent neural networks on gpus. *CoRR*, abs/1604.01946, 2016.

[12] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. Eliminating redundant fragment shader executions on a mobile gpu via hardware memoization. *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 529–540, 2014.

[13] Pinkesh Badjatiya, Shashank Gupta, Manish Gupta, and Vasudeva Varma. Deep learning for hate speech detection in tweets. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 759–760, 2017.

[14] Pierre Baldi, Søren Brunak, Paolo Frasconi, Gianluca Pollastri, and Giovanni Soda. Bidirectional dynamics for protein secondary structure prediction. In *Sequence Learning*, pages 80–104. Springer, 2001.

[15] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[16] Ond rej Bojar, Rajen Chatterjee, Christian Federmann, Barry Haddow, Matthias Huck, Chris Hokamp, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Matt Post, Carolina Scarton, Lucia Specia, and Marco Turchi. Findings of the 2015 workshop on statistical machine translation. In *Proceedings of the Tenth Workshop on Statistical Machine Translation*, pages 1–46, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[17] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on fpga. *arXiv preprint arXiv:1511.05552*, 2015.

[18] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274, 2015.

[19] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 367–379, Piscataway, NJ, USA, 2016. IEEE Press.

[20] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[21] François Chollet et al. Keras. https://github.com/fchollet/keras, 2015.

[22] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[23] Andrew M. Dai and Quoc V. Le. Semi-supervised sequence learning. *CoRR*, abs/1511.01432, 2015.

[24] Gregory Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. Persistent rnns: Stashing recurrent weights on-chip. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, page 2024–2033. JMLR.org, 2016.

[25] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, et al. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 395–408, 2017.

[26] Ronen Eldan and Ohad Shamir. The power of depth for feedforward neural networks. *Conference on Learning Theory*, 12 2015.

[27] Asbjørn Følstad and Petter Bae Brandtzæg. Chatbots and the new world of hci. *interactions*, 24(4):38–42, 2017.

[28] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, et al. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 1–14. IEEE Press, 2018.

[29] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. A configurable cloud-scale dnn processor for real-time ai. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, pages 1–14, Piscataway, NJ, USA, 2018. IEEE Press.

[30] Merlin Friesen. Linux power management optimization on the nvidia jetson platform. https://events.static.linuxfound.org/sites/events/files/slides/Linux_Low_Power_ELC_SanDiego.pdf.

[31] Chang Gao, Daniel Neil, Enea Ceolini, Shih-Chii Liu, and Tobi Delbruck. Deltarnn: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 21–30, New York, NY, USA, 2018. Association for Computing Machinery.

[32] Chang Gao, Antonio Rios-Navarro, Xi Chen, Tobi Delbruck, and Shih-Chii Liu. Edgedrnn: Enabling low-latency recurrent neural network edge inference. *arXiv preprint arXiv:1912.12193*, 2019.

[33] Pin Gao, Lingfan Yu, Yongwei Wu, and Jinyang Li. Low latency rnn inference with cellular batching. In *Proceedings of the Thirteenth EuroSys Conference*, EuroSys '18, pages 31:1–31:15, New York, NY, USA, 2018. ACM.

[34] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. NASA STI/Recon Technical Report N, February 1993.

[35] Antonio González, Jordi Tubella, and Carlos Molina-Clemente. Trace-Level Reuse. In *ICPP*, pages 30–, 1999.

[36] https://en.wikipedia.org/wiki/Google_Assistant.

[37] https://www.nextplatform.com/2018/05/10/tearing-apart-googles-tpu-3-0-ai-coprocessor/.

[38] https://coral.ai/docs/edgetpu/benchmarks/.

[39] https://en.wikipedia.org/wiki/Grammarly.

[40] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm networks. In *Neural Networks, 2005. IJCNN'05. Proceedings. 2005 IEEE International Joint Conference on*, volume 4, pages 2047–2052. IEEE, 2005.

[41] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. Fpga-based accelerator for long short-term memory recurrent neural networks. In *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*, pages 629–634. IEEE, 2017.

[42] Udit Gupta, Brandon Reagen, Lillian Pentecost, Marco Donato, Thierry Tambe, Alexander M. Rush, Gu-Yeon Wei, and David M. Brooks. Masr: A modular accelerator for sparse rnns. *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–14, 2019.

[43] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. Ese: Efficient speech recognition engine with sparse lstm on fpga. In *FPGA*, pages 75–84, 2017.

[44] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[45] Song Han, Huizi Mao, and William J. Dally. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding. In Yoshua Bengio and Yann LeCun, editors, *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

[46] Simon Haykin. *Neural Networks: A Comprehensive Foundation (3rd Edition)*. Prentice-Hall, Inc., USA, 2007.

[47] Robert Hecht-Nielsen. Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier, 1992.

[48] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[49] Connor Holmes, Daniel Mawhirter, Yuxiong He, Feng Yan, and Bo Wu. Grnn: Low-latency and scalable rnn inference on gpus. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, pages 41:1–41:16, New York, NY, USA, 2019. ACM.

[50] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

[51] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, pages 1–12, New York, NY, USA, 2017. ACM.

[52] Patrick Judd, Jorge Albericio, Tayler Hetherington, Tor M. Aamodt, and Andreas Moshovos. Stripes: Bit-serial deep neural network computing. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, pages 19:1–19:12, Piscataway, NJ, USA, 2016. IEEE Press.

[53] Patrick Judd, Jorge Albericio, Tayler H. Hetherington, Tor M. Aamodt, Natalie D. Enright Jerger, Raquel Urtasun, and Andreas Moshovos. Reduced-precision strategies for bounded memory in deep neural nets. *CoRR*, abs/1511.05236, 2015.

[54] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015.

[55] Viacheslav Khomenko, Oleg Shyshkov, Olga Radyvonenko, and Kostiantyn Bokhan. Accelerating recurrent neural network training using sequence bucketing and multi-gpu data parallelization. *CoRR*, abs/1708.05604, 2017.

[56] Minje Kim and Paris Smaragdis. Bitwise neural networks. *arXiv preprint arXiv:1601.06071*, 2016.

[57] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[58] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander M Rush. Opennmt: Open-source toolkit for neural machine translation. *arXiv preprint arXiv:1701.02810*, 2017.

[59] Richard E. Korf. Multi-way number partitioning. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 538–543, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[60] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[61] Minjae Lee, Kyuyeon Hwang, Jinhwan Park, Sungwook Choi, Sungho Shin, and Wonyong Sung. Fpga-based low-power speech recognition with recurrent neural networks. In *Signal Processing Systems (SiPS), 2016 IEEE International Workshop on*, pages 230–235. IEEE, 2016.

[62] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. Fpga acceleration of recurrent neural network based language model. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 111–118. IEEE, 2015.

[63] Zhuohan Li, Di He, Fei Tian, Wei Chen, Tao Qin, Liwei Wang, and Tie-Yan Liu. Towards binary-valued gates for robust lstm training, 2018.

[64] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[65] R. P. Lippmann. An introduction to computing with neural nets. *IEEE ASSP Magazine*, pages 4–22, April 1987.

[66] Zachary Chase Lipton. A critical review of recurrent neural networks for sequence learning. *CoRR*, abs/1506.00019, 2015.

[67] Xiaodong Liu, Jianfeng Gao, Xiaodong He, Li Deng, Kevin Duh, and Ye-Yi Wang. Representation learning using multi-task deep neural networks for semantic classification and information retrieval. 2015.

[68] Gerald M Maggiora, David W Elrod, and Robert G Trenary. Computational neural networks as model-free mapping devices. *Journal of chemical information and computer sciences*, 32(6):732–741, 1992.

[69] Yajie Miao, Mohammad Gowayyed, and Florian Metze. Eesen: End-to-end speech recognition using deep rnn models and wfst-based decoding. In *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*, pages 167–174. IEEE, 2015.

[70] Micron Inc. TN-53-01: LPDDR4 System Power Calculator. https://www.micron.com/support/tools-and-utilities/power-calc.

[71] Naveen Muralimanohar, Rajeev Balasubramonian, and Norman P Jouppi. Cacti 6.0: A tool to model large caches. *HP Laboratories*, pages 22–31, 2009.

[72] NVIDIA. NVIDIA TEGRA X1 new mobile superchip. http://international.download.nvidia.com/pdf/tegra/Tegra-X1-whitepaper-v1.0.pdf.

[73] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. Librispeech: an asr corpus based on public domain audio books. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5206–5210. IEEE, 2015.

[74] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[75] Eunhyeok Park, Dongyoung Kim, and Sungjoo Yoo. Energy-efficient neural network accelerator based on outlier-aware low-precision computation. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, ISCA '18, page 688–698. IEEE Press, 2018.

[76] Seung-min Park, Daeyoun D Won, Brian J Lee, Diego Escobedo, Andre Esteva, Amin Aalipour, Jessie Ge T, Jung Ha Kim, Susie Suh, Elliot H Choi, Alexander X Lozano, Chengyang Yao, Sunil Bodapati, Friso B Achterberg, Jeesu Kim, Hwan Park, Youngjae Choi, Woo Jin Kim, Jung Ho Yu, Alexander M Bhatt, Jong Kyun Lee, Ryan Spitler, Shan X Wang, and Sanjiv S Gambhir. A mountable toilet system for personalized health monitoring via the analysis of excreta. *Nature Biomedical Engineering*, pages 1–12, 2020.

[77] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[78] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*, 2017.

[79] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, pages 8024–8035, 2019.

[80] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer, 2016.

[81] M. Riera, J. Arnau, and A. González. Cgpa: Coarse-grained pruning of activations for energy-efficient rnn inference. *IEEE Micro*, 39(5):36–45, Sep. 2019.

[82] Marc Riera, Jose-Maria Arnau, and Antonio González. Computation reuse in dnns by exploiting input similarity. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*, pages 57–68. IEEE Press, 2018.

[83] Anthony Rousseau, Paul Deléglise, and Yannick Esteve. Ted-lium: an automatic speech recognition dedicated corpus. In *LREC*, pages 125–129, 2012.

[84] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[85] Christopher M. Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, page 265–278, New York, NY, USA, 2006. Association for Computing Machinery.

[86] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. A systematic methodology for characterizing scalability of dnn accelerators using scale-sim.

[87] Ananda Samajdar, Yuhao Zhu, Paul Whatmough, Matthew Mattina, and Tushar Krishna. Scale-sim: Systolic cnn accelerator simulator. *arXiv preprint arXiv:1811.02883*, 2018.

[88] Murat Hüsnü Sazlı. A brief review of feed-forward neural networks. 2006.

[89] Mike Schuster and Kuldip K Paliwal. Bidirectional recurrent neural networks. *IEEE Transactions on Signal Processing*, 45(11):2673–2681, 1997.

[90] Frank Seide and Amit Agarwal. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, pages 2135–2135, New York, NY, USA, 2016. ACM.

[91] Mohammad Javad Shafiee, Brendan Chywl, Francis Li, and Alexander Wong. Fast yolo: A fast you only look once system for real-time embedded object detection in video. *arXiv preprint arXiv:1709.05943*, 2017.

[92] H. Sharma, J. Park, N. Suda, L. Lai, B. Chau, V. Chandra, and H. Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775, June 2018.

[93] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio González. E-pur: An energy-efficient processing unit for recurrent neural networks. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, PACT '18, pages 18:1–18:12, New York, NY, USA, 2018. ACM.

[94] Franyell Silfa, Gem Dot, Jose-Maria Arnau, and Antonio González. Neuron-level fuzzy memoization in rnns. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 782–793, New York, NY, USA, 2019. Association for Computing Machinery.

[95] Avinash Sodani and Gurindar S. Sohi. Dynamic instruction reuse. ISCA '97, pages 194–205, 1997.

[96] V Subramaniyaswamy, R Logesh, M Abejith, Sunil Umasankar, and A Umamakeswari. Sentiment analysis of tweets for estimating criticality and security of events. In *Improving the Safety and Efficiency of Emergency Services: Emerging Tools and Technologies for First Responders*, pages 293–319. IGI Global, 2020.

[97] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[98] Daniel Svozil, Vladimir Kvasnicka, and Jiri Pospichal. Introduction to multi-layer feed-forward neural networks. *Chemometrics and intelligent laboratory systems*, 39(1):43–62, 1997.

[99] Synopsys. https://www.synopsys.com/.

[100] Jie Tang, Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. Enabling deep learning on iot devices. *Computer*, 50(10):92–96, 2017.

[101] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1653–1660, 2014.

[102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[103] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, Laurent Sifre, Trevor Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 575(7782):350–354, Nov 2019.

[104] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: Lessons learned from the 2015 MSCOCO image captioning challenge. *CoRR*, abs/1609.06647, 2016.

[105] Guibin Wang, YiSong Lin, and Wei Yi. Kernel fusion: An effective method for better power efficiency on multithreaded gpu. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 344–350. IEEE Computer Society, 2010.

[106] Peiqi Wang, Xinfeng Xie, Lei Deng, Guoqi Li, Dongsheng Wang, and Yuan Xie. Hitnet: Hybrid ternary recurrent neural network. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 602–612, Red Hook, NY, USA, 2018. Curran Associates Inc.

[107] Shuo Wang, Zhe Li, Caiwen Ding, Bo Yuan, Qinru Qiu, Yanzhi Wang, and Yun Liang. C-lstm: Enabling efficient lstm using structured compression techniques on fpgas. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18, page 11–20, New York, NY, USA, 2018. Association for Computing Machinery.

[108] G. M. Weiss, J. L. Timko, C. M. Gallagher, K. Yoneda, and A. J. Schreiber. Smartwatch-based activity recognition: A machine learning approach. In *2016 IEEE-EMBS International Conference on Biomedical and Health Informatics (BHI)*, pages 426–429, 2016.

[109] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[110] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[111] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. PASTE '07, pages 75–82, 2007.

[112] Reza Yazdani, Olatunji Ruwase, Minjia Zhang, Yuxiong He, Jose-Maria Arnau, and Antonio González. Lstm-sharp: An adaptable, energy-efficient hardware accelerator for long short-term memory, 2019.

[113] Joe Yue-Hei Ng, Matthew Hausknecht, Sudheendra Vijayanarasimhan, Oriol Vinyals, Rajat Monga, and George Toderici. Beyond short snippets: Deep networks for video classification. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[114] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. Deepcpu: Serving rnn-based deep learning models 10x faster. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 951–965, Boston, MA, July 2018. USENIX Association.

[115] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: An approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 701–706, San Jose, CA, USA, 2015. EDA Consortium.

[116] Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, and Qiang Xu. Approxann: An approximate computing framework for artificial neural network. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, pages 701–706, San Jose, CA, USA, 2015. EDA Consortium.

[117] Xiaofan Zhang, Xinheng Liu, Anand Ramachandran, Chuanhao Zhuge, Shibin Tang, Peng Ouyang, Zuofu Cheng, Kyle Rupnow, and Deming Chen. High-performance video content recognition with long-term recurrent convolutional network for fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2017.

Glossary

WER (Word Error Rate). Number of insertions plus deletions plus substitutions that are required to convert the recognized word sequence into the reference word sequence, divided by the total number of words of the reference utterance.

Real-Time-Factor (Real Time Factor). It is the ratio between the amount of time required to decode an audio fragment and the length of the fragment. For example, a real-time factor of 0.7 xRT means that each second of audio requires 0.7 seconds to decode (lower RTF means faster decoding).

Bleu (Bleu). It is a metric for evaluating a generated sentence to a reference sentence. It works by counting matching n-grams in the candidate translation to n-grams in the reference text. The larger the Bleu the better.

Inference (Inference). In the context of machine learning, it refers to using a trained model to make a prediction.

Deep-Neural-Network (DNN). A machine-learning algorithm which includes one input and one output layer of neurons plus several hidden layers. It is trained with a large amount of data to find the correct mathematical representation to compute the network output from a given input.

Recurrent-Neural-Network (RNN). A machine-learning primitive consisting of several layers composed of one RNN cell. Each RNN cell recurrently processes an input sequence while storing information from past evaluations. Then, this information is used to compute the next output of the network.

DNN-Accelerator (DNN Accelerator). A class of specialized computer system designed to improve the performance and energy efficiency of DNN algorithms.

Sentiment-Analysis(Sentiment Analysis). It refers to the classification of emotions (positive, negative, and neutral) within text data using text analysis techniques such as natural language processing.

cuDNN (cuDNN). A GPU-optimized library of primitives for deep neural networks.

cuBlas (cuBlas). A GPU-accelerated implementation of the basic linear algebra subroutines (BLAS).

Qm.n (Qm.n). It is a fixed-point format to represent decimal numbers. The m represents the bit-width of the integer part, whereas the n represents the bit-width of the fractional part.

Batch (Batch). It is a group of input samples that are processed concurrently either during inference or training.

Batch Size (Batch Size). It is the number of samples in a batch.

Verilog (Verilog). It is a Hardware Description Language for describing electronic circuits and systems.

CACTI (CACTI). It is an integrated model for cache and memory access time, cycle time, area, leakage, and dynamic power.

Output-Stationary-Dataflow (Output Stationary Dataflow). It is a dataflow where the accumulation of partial sums for the same output activation value is kept local in a register file. In order to keep the accumulation of each partial sum stationary in the register file, input activations are streamed across the processing elements.

Neuron (Neuron). It is the basic computation unit of neural networks. Its output is computed as the dot product between the input and synaptic weights.

Convolutional-Neural-Networks (Convolutional-Neural-Networks). They are DNNs that employ convolution operations instead of the general matrix multiplication in at least one of their layers. CNNs are normally employed for image classification.

Synaptic-Weights (Synaptic Weights). They represent the connections between neurons. Typically, they determine the relevance of the information passed by a neuron to the network input and/or with other neurons.