*electronics*

MDPI

*Article*

# Improving Performance Estimation for Design Space Exploration for Convolutional Neural Network Accelerators †

**Martin Ferianc [1,*,‡], Hongxiang Fan [2,‡], Divyansh Manocha [3], Hongyu Zhou [4], Shuanglong Liu [5], Xinyu Niu [6] and Wayne Luk [2]**

1 Department of Electronic and Electrical Engineering, University College London, London WC1E 7JE, UK
2 Department of Computing, Imperial College London, London SW7 2AZ, UK; h.fan17@imperial.ac.uk (H.F.); w.luk@imperial.ac.uk (W.L.)
3 Independent Researcher, Cambridge CB23 7UE, UK; divyanshmanocha@gmail.com
4 Independent Researcher, Changsha 410081, China; hongyu.hyzhou@gmail.com
5 School of Physics and Electronics, Hunan Normal University, Changsha 410081, China; liu.shuanglong@hunnu.edu.cn
6 Corerain Technologies Ltd., Shanghai 201203, China; xinyu.niu@corerain.com
* Correspondence: martin.ferianc.19@ucl.ac.uk
† This paper is an extended version of our paper published in Lecture Notes in Computer Science, vol. 12083. Springer, Cham.
‡ These authors contributed equally to this work.

**Abstract:** Contemporary advances in neural networks (NNs) have demonstrated their potential in different applications such as in image classification, object detection or natural language processing. In particular, reconfigurable accelerators have been widely used for the acceleration of NNs due to their reconfigurability and efficiency in specific application instances. To determine the configuration of the accelerator, it is necessary to conduct design space exploration to optimize the performance. However, the process of design space exploration is time consuming because of the slow performance evaluation for different configurations. Therefore, there is a demand for an accurate and fast performance prediction method to speed up design space exploration. This work introduces a novel method for fast and accurate estimation of different metrics that are of importance when performing design space exploration. The method is based on a Gaussian process regression model parametrised by the features of the accelerator and the target NN to be accelerated. We evaluate the proposed method together with other popular machine learning based methods in estimating the latency and energy consumption of our implemented accelerator on two different hardware platforms targeting convolutional neural networks. We demonstrate improvements in estimation accuracy, without the need for significant implementation effort or tuning.

**Keywords:** field-programmable gate array; deep learning; neural network; performance estimation; Gaussian process

check for updates

**Citation:** Ferianc, M.; Fan, H.; Manocha, D.; Zhou, H.; Liu, S.; Niu, X.; Luk, W. Improving Performance Estimation for Design Space Exploration for Convolutional Neural Network Accelerators. *Electronics* **2021**, *10*, 520. https://doi.org/10.3390/electronics10040520

Academic Editor: Alexander Barkalov

Received: 28 December 2020
Accepted: 10 February 2021
Published: 23 February 2021

**Publisher's Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

**Copyright:** © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

## 1. Introduction

Recently, neural networks (NNs) have demonstrated superhuman performance in a multitude of tasks, such as in image classification [1], object detection [2], semantic segmentation [3] or natural language processing [4]. NNs are also making their way into real-life practical applications, such as in medical diagnostics [5], autonomous driving [6] or aviation [7–9]. While in medicine, the applications of NNs are primarily limited by their algorithmic performance, in other practical scenarios such as in autonomous driving, their hardware performance needs to also be considered in addition to their decision making capabilities. The hardware performance is usually considered in terms of latency or energy efficiency, which is especially crucial when aiming at real-time response rates. While it is indeed possible to run NNs on stock hardware platforms such as central processing units (CPUs) or graphical processing units (GPUs), to achieve peak hardware performance, it is

also necessary to consider reconfigurable hardware accelerators [10]. Considering the rapid pace of NN architecture design, accelerators need to be partially reconfigurable such that they are adaptable to the new generation of NN designs, while still achieving favourable hardware performance.

Therefore, to fully utilise the performance capabilities of a reconfigurable accelerator, it is necessary to perform design space exploration (DSE) [11] to determine the optimal hardware configuration of the accelerator, given the desired NN architectures. The search space when performing DSE is determined by the available accelerator's configuration domains which can, for example, be determined by the levels of implementable parallelism [10]. Naively, DSE is conducted by systematically synthesising different configurations of a given accelerator on the hardware platform and measuring the real-world performance of the desired NNs on the accelerator. Given a large search space, consisting of different configurations of the accelerator, the time and resource costs of actually implementing the accelerator on the target hardware platform limit the speed of DSE. Practically, it is therefore necessary to accurately estimate the hardware performance during DSE with respect to multiple different hardware specifications, to enable the fast exploration and exploitation of the available configurations for the given NNs.

There are several performance estimation frameworks for reconfigurable accelerators [12–14]; however, estimating the performance without knowing the run-time intricacies when running different NNs is still a challenging task. There are two main reasons for this complication: (1) the cost of executing a certain operation on hardware varies by on/off-chip communication, synchronisation, control signals, I/O interruptions, in particular for the NN accelerators, the NN's architecture, complicating the estimation; (2) it is difficult to accurately select the most representative design features for all hardware specifications during performance estimation.

In this work, we propose a novel approach for performance estimation of custom convolutional neural network (CNN) accelerators. The proposed method constitutes a Gaussian process regression model [15] coupled with features that can be readily read off datasheets for the underlying hardware platform or the target algorithm (a tutorial code is available at https://git.io/Jv31c). We evaluate the method for estimating layer-wise latency, as well as network-wise latency and energy consumption. Experiments were conducted with respect to two hardware platforms, the Intel Arria GX 1150 field-programmable gate array (FPGA), as well as a structured application-specific integrated circuit (ASIC) implementation of the targeted accelerator. We compared the proposed approach to other machine learning-inspired methods such as linear regression (LR), gradient tree boosting (GTB) or a feed-forward fully-connected NN. The proposed approach is simple to implement, fast in providing predictions and more accurate in comparison to the other compared methods in estimating both latency and energy. This article extends our previous work [16] by further evaluation with respect to estimating an additional hardware metric, energy consumption, by benchmarking the proposed method with respect to an additional hardware implementation platform (ASIC) and by supportive software experiments. The further experimentation proves that the Gaussian process is an accurate estimator that can be used to estimate the hardware performance for running CNNs.

In Section 2, we discuss the background on NN design and the related work on performance estimation. Then, in Section 3, we introduce the proposed method, followed by Section 4, where we describe the implemented hardware design of the benchmarked accelerator. Then, we present the experiments, results and discussion in Section 5. Lastly, we conclude the work in Section 6.

## 2. Background and Related Work

In this section, we present an overview of NNs and their compute pattern and related work on performance estimation methods.

### 2.1. Neural Networks

NNs are built by stacking several mathematical operations on top of each other, otherwise known as layers. In this work, we mainly demonstrate our method on an accelerator for CNNs; however, the proposed method is not limited to accelerators for CNNs. The processing of a CNN is usually done in a layer-by-layer fashion; nevertheless, most modern networks [17–19] have residual or concatenative connections between them [17]. Specifically for CNNs, frequently used layers are 2D convolutional, fully-connected or pooling layers interchanged with element-wise applied non-linearities [20]. Convolutional or fully-connected layers aim to learn useful features that can be used to recognise patterns in the input data, while pooling aims to reduce the representation and pool the most important information, while processing the data through the NN. Practically, convolutional and fully-connected layers take up over 90% of the computation and energy consumption in a CNN model [2,21,22]. The algorithm behind 2D convolution is shown in Algorithm 1. The notation used in this paper is presented in Table 1.

**Algorithm 1** Convolution.

**Input**: Input feature map **I** of shape $C \times H_I \times W_I$; weight matrix **W** of shape $F \times C \times K \times K$

**Output**: Output feature map **O** of shape $F \times H_O \times W_O$

1: for $(f = 0; f < F; f++)$
2: for $(c = 0; c < C; c++)$
3: for $(h = 0; h < H_O; h++)$
4: for $(w = 0; w < W_O; w++)$
5: $\mathbf{O}[f][h][w] \mathrel{+}= \sum_{i=1}^{K-1} \sum_{j=1}^{K-1} \mathbf{W}[f][c][i][j] * \mathbf{I}[c][h*s+i][w*s+j]$

**Table 1.** Notation used in this paper.

| | | | |
|---|---|---|---|
| $H_I$ | Height of the input feature map | $W_I$ | Width of the input feature map |
| $H_O$ | Height of the output feature map | $W_O$ | Width of the output feature map |
| $K$ | Kernel size | $F$ | Number of filters |
| $C$ | Number of channels | $s$ | Stride in a convolution |
| $W$ | Weights in a neural network | $PF$ | Parallelism in the filter dimension |
| $PC$ | Parallelism in the channel dimension | $PV$ | Parallelism in the data vector dimension |
| $M_{CLK}$ (MHz) | Memory access clock cycle time | $L_{CLK}$ (MHz) | Logic clock cycle time |
| $M_{EFF}$ (%) | Memory transfer efficiency | $S$ (bits) | Memory transfer size |
| $DW$ (bits) | Processing data width | $M$ | Number of input features |
| $B$ | Number of layers in a neural network | $N$ | Number of training samples |

As illustrated in Algorithm 1, the convolution accepts a $C \times H_I \times W_I$ sized input feature map, and then, the input is convolved with a kernel with the shape of $F \times C \times K \times K$. Each kernel window with the size of $K \times K$ is applied to one channel of the input $H_I \times W_I$ by sliding the kernel with a stride of $s$ to produce one output feature map $H_O \times W_O$; then, the results of $C$ channels are accumulated to produce one filter of the output. All filters of the output feature maps $F \times H_O \times W_O$ are generated by repeating this process $F$ times. A fully-connected layer can be re-interpreted as a convolution by considering the kernel size $K = 1$. Utilizing this compute pattern, it is then possible to summarize the number of compute operations, as well as the number of memory transfers, as shown in Table 2. At the same time, given the different for-loops in Algorithm 1, it is possible to parallelise the convolution operation in each for-loop dimension: filter, channel, data vector or kernel. In Section 4, we introduce the implemented accelerator, which is capable of taking advantage of this property in multiple dimensions.

**Table 2.** Number of operations and the data size for a convolution.

| Sizes | Number of Operations/Data Size |
|---|---|
| Number of compute operations | $F \times C \times H_I \times W_I \times K \times K$ |
| Input size | $H_I \times W_I \times C$ |
| Weights size | $F \times C \times K \times K$ |
| Output size | $H_O \times W_O \times F$ |

### *2.2. Performance Estimation*

As discussed in Section 1, the most accurate and reliable method for determining the performance of a CNN for a specific system configuration is deploying the CNN on the hardware platform and measuring its performance. A significant drawback of this method is that it requires re-implementation for different hardware specifications on the hardware's fabric. Given a large number of potential configurations that might need to be benchmarked during DSE, this approach is too time consuming and resource demanding. Therefore, it is more feasible and practical to perform DSE with respect to an estimate of the performance at the software level, rather than running the CNN for each hardware configuration of different hardware architectures. Considering a complex accelerator for multi-layer CNNs, it is likely that due to the intricacy of the data manipulation or the compute, the performance for the CNNs will need to be estimated on a case-by-case basis. Therefore, this approach is infeasible in general, as it is usually constrained to a single hardware configuration. Nevertheless, there have been a few researchers who have proposed general performance estimation methodologies [12–14].

A performance estimation framework for reconfigurable dataflow platforms was proposed by Yasudo et al. [12], which can analytically determine the number of accelerator units suitable for an application. Dai et al. [13] proposed an estimation method based on a GTB and a high-level synthesis report. However, their method requires a significant amount of data and features from the synthesis report, which might not be available, especially when high-level synthesis is not being used to implement the accelerator. Liu et al. proposed a general heuristic based method [14] for estimating the performance of FPGA based CNN accelerators and that is now used as the standard go-to estimation method. The heuristic analytic approach does not depend on any potentially collected measurements to perform the estimation, and it is simple to implement since it relies only on the variables that can be easily read from the respective datasheets for the hardware platform or the algorithmic configuration. Nevertheless, this general estimation method usually computes the most optimistic estimate, and it does not take into account communication, synchronisation or control. One way to refine the estimation is that we can collect a few runtime data points and use them to improve the estimate.

Therefore, in our work, we propose using a Gaussian process (GP) regression model [23] together with data samples collected by running the CNN on real hardware. GP is a model built on Bayesian probabilistic theory, which can embody prior knowledge into the predictive model and can be used for the regression of real-valued non-linear targets [23].

## 3. Method

In this section, we motivate and describe the proposed method for performance estimation, which is based on a GP regression model.

Given a dataset $\mathcal{D} = \{(\boldsymbol{x}_i, y_i)\}; i = 1, \ldots, N$ consisting of $N$ observations with inputs and outputs as $\boldsymbol{x}_i \in \mathbb{R}^M$ and $y_i \in \mathbb{R}^1$, respectively, a function $f$ needs to be induced to hypothesise $\boldsymbol{y}_*$ on new, previously unseen, inputs $\boldsymbol{x}_*$. $\boldsymbol{x}$ represents a vector of $M$ features, while $y$ represents the real-valued target that is to be estimated in this case. As discussed in the previous Section 2.2, there are multiple function classes that can be used to perform this task.

A naive parametric approach would make use of a predictive conditional distribution that can be written as $p(y_*|\boldsymbol{w}, \mathcal{D}, \boldsymbol{x}_*)$. This approach constitutes an LR, using parameters

$\boldsymbol{w}$, such that the prediction is made as $y = \sum_m^M w_m x_m$. It requires learning the parameters $\boldsymbol{w}$, which represent one potential function realisation $f$ that fits the data.

Assuming a Gaussian weight prior $p(\boldsymbol{w}) = \mathcal{N}(\boldsymbol{w}|\boldsymbol{0}, \boldsymbol{\Sigma}_w)$, with some pre-defined covariance matrix $\boldsymbol{\Sigma}_w$, we can induce a Gaussian distribution on any set of $\boldsymbol{y}$: $p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}|\boldsymbol{\mu}, \boldsymbol{K})$, where $\boldsymbol{K} \in \mathbb{R}^{N\times N}$ is the covariance matrix characterised by a covariance function and $\boldsymbol{\mu}$ represents the mean. This leads to the consideration of a non-parametric predictor, where instead of learning $\boldsymbol{w}$, the focus is shifted towards inferring an entire distribution of function classes for explaining the data. Specifically, a non-parametric predictor uses a parametric model and integrates the parameters. A prior $p(\theta)$ induces a distribution over plausible functions, where $\theta$ is a latent random variable. Using such a probabilistic modelling framework, we can sample plausible data-fitting functions directly. This approach avoids necessitating a decision on which predefined class of function predictors to use, as it considers all of them. The assumption that any set of values specified at an arbitrary point $\mathbf{x}_i$ over functions is Gaussian distributed leads to a GP model.

GP is a flexible Bayesian model characterised by a finite collection of Gaussian random variables $[f_1, f_2, \ldots]$, such that for any finite set of plausible inputs $\boldsymbol{X}_*$, the vector $\boldsymbol{f}_* = f(\boldsymbol{X}_*)$ follows a Gaussian distribution [23]. The stochastic process can be entirely determined by second-order statistics: a mean function $m(.)$ and a kernel (covariance) function $k(.,.)$. The mean function represents the value that the mean across the functions $\boldsymbol{f}$ tends towards. The covariance matrix $\boldsymbol{K}$ is characterised by the kernel function values $[\boldsymbol{K}]_{i,j} = k(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{\phi}(\boldsymbol{x}_i)^T \boldsymbol{\phi}(\boldsymbol{x}_j)$, for some non-linear function $\boldsymbol{\phi}(.)$, which represent the value that the sample covariance for all sampled functions tends towards for the points $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. The kernel encodes structural information of the latent function $f$ and must be symmetric and positive semi-definite.

For $N$ Gaussian observations $\boldsymbol{X}_N \in \mathbb{R}^{N\times M}$; $\boldsymbol{Y}_N \in \mathbb{R}^{N\times 1}$, $y_i = f(\boldsymbol{x}_i) + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(\epsilon_i|0, \sigma^2)$, the posterior for unseen data $\boldsymbol{X}_*$ is defined as in Equations (1) and (2) (for a detailed derivation, please refer to [23]):

$$\boldsymbol{f}_*|\boldsymbol{y} \sim \mathcal{N}(\boldsymbol{m}_{*|N}, \boldsymbol{K}_{*,*|N}) \tag{1}$$

$$\begin{aligned} \boldsymbol{m}_{*|N} &= m(\boldsymbol{X}_N) + \boldsymbol{K}_{*,N}(\boldsymbol{K}_{N,N} + \sigma^2 \boldsymbol{I})^{-1}(\boldsymbol{Y}_N - m(\boldsymbol{X}_N)) \\ \boldsymbol{K}_{*,*|N} &= \boldsymbol{K}_{*,*} - \boldsymbol{K}_{*,N}(\boldsymbol{K}_{N,N} + \sigma^2 \boldsymbol{I})^{-1}\boldsymbol{K}_{N,*} \end{aligned} \tag{2}$$

Furthermore, training the GP requires finding appropriate latent random variables or hyperparameters $\boldsymbol{\theta}$. Considering the posterior over hyperparameters: $p(\boldsymbol{\theta}|\boldsymbol{X}, \boldsymbol{y}) = \frac{p(\boldsymbol{y}|\boldsymbol{X},\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(\boldsymbol{y}|\boldsymbol{X})}$, hyperparameters $\boldsymbol{\theta}^*$ are obtained through maximising the log of marginal likelihood $\boldsymbol{\theta}^* = \arg\max_\theta \log p(\boldsymbol{y}|\boldsymbol{X}, \boldsymbol{\theta}) + \log p(\boldsymbol{\theta})$.

In this paper, we propose to use a GP regression model as outlined above to predict the performance of an algorithm realisation on a given accelerator and a hardware platform. We propose to use the characteristics of the accelerator at design time and the target NN as features, as shown in Table 1, with respect to which we can predict the target performance measure (a tutorial code is available at https://git.io/Jv31c). Practically, this means that an input vector $\boldsymbol{x}$ is a vector of $M$ features with algorithmic or hardware properties for one configuration of the system, while $y$ can represent the performance that is to be estimated. The features of the input vector $\boldsymbol{x}$ being used are those that are already known and used in the standard analytic estimation [14], avoiding the need for any additional feature extraction from the dataset or the datasheets. These features consist of characteristics of the CNN to be run, as well as the hardware accelerator. Additionally, it is possible to embody the standard analytic method into the GP based estimator, through using it as the mean function $m(.)$. This model enables us to use any available measurements as training data and does not restrict us to one class of predictors; it considers a plausible family of best fitting models that are characterised by the kernel and the mean function. The proposed method is able to make predictions outside of the observed data samples without collapsing [23]. At the same time, by choosing the features given by the datasheets, the

model is more interpretable than an NN or an LR, where the corresponding uninterpretable weights $\boldsymbol{w}$ need to be learned. Moreover, the Gaussian noise assumption can be interpreted as an additive instrumentation error, while collecting measurements. Furthermore, if used during DSE, the GP model can additionally provide an uncertainty estimate for its predictions, which can more precisely guide the exploration and the exploitation of the search space [23]. The overall system diagram, including all the necessary parts of the prediction methodology, is presented in Figure 1. The dashed lines symbolise the fitting of the GP, through providing hardware measurements, along with the characteristic NN and hardware features, to the GP to obtain the $\boldsymbol{\theta}^*, \boldsymbol{Y}_N, \boldsymbol{K}_{N,N}$ to be used during the evaluation. During the evaluation, the features and the fitted GP model are then used for prediction.

For a training set of size $N$ samples, the computational complexity of the training scales in $\sim\mathcal{O}(N^3)$ due to the unavoidable Cholesky factorisation, while the prediction is $\sim\mathcal{O}(N^2)$, and the memory requirements are $\sim\mathcal{O}(NM + N^2)$. Therefore, given a typical number of collected real-world measurements (which is $<$1000) for different configurations of the accelerator, the method is scalable to be used in practice.

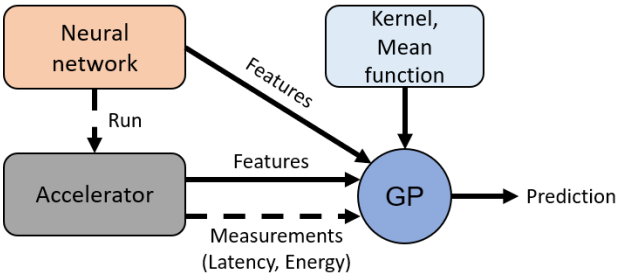


**Figure 1.** Overview of the proposed prediction methodology based on a Gaussian process (GP).

In the next section, we present the CNN accelerator on which we used the proposed method. We compare our approach with other estimators in predicting layer-wise latency and network-wise latency and energy consumption.

## 4. Hardware Design

In this section, we detail the accelerator architecture, the performance for multiple different CNN architectures of which we aim to estimate.

### *4.1. Accelerator's Architecture*

The hardware design of our accelerator is illustrated in Figure 2. The design consists of a CNN engine, a central communication interconnect and an off-chip main memory. The weights of the whole network are transferred and stored in the off-chip memory via a central communication interconnect before the processing. The CNN engine is composed of an input buffer, a weight buffer, a convolutional processing engine (PE) and other functional modules including batch normalisation (BN) [24], shortcut (SC) [17], pooling (Pool) and rectified linear unit (ReLU) activation. In order to fully utilise the extensive concurrency exhibited in CNNs and improve the hardware efficiency, we support three types of fine-grained parallelism in our CNN engine: filter parallelism (PF), channel parallelism (PC) and vector parallelism (PV). The accelerator processes each layer in a CNN one-by-one, and the intermediate results between layers are transferred and stored in the off-chip memory, in case the output size is bigger than the available on-chip memory. To achieve higher hardware performance, the accelerator is designed to support 8 bit operations.

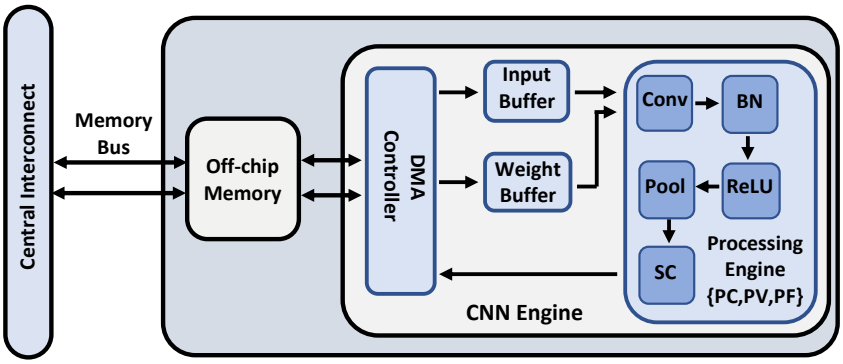


**Figure 2.** The convolutional neural network accelerator's design. SC, shortcut; PC, channel parallelism; PV, vector parallelism; PF, filter parallelism; DMA, direct memory access.

To avoid large memory consumption on the on-chip memory, we adopt the channel-major computational pattern for convolution, which is illustrated in Algorithm 2. In our channel-major PE, the computation required along the channel dimension in each filter is finished first. In this way, the on-chip memory only needs to cache the intermediate results for one filter, which largely decreases the memory usage.

In this paper, we used this accelerator design to perform the benchmarking of our proposed estimator method in estimating layer-wise latency, network-wise latency and energy consumption.

**Algorithm 2** Channel-major computational pattern.

**Input**: Input feature map **I** of shape $C \times H_I \times W_I$; weight matrix **W** of shape $F \times C \times K \times K$

**Output**: Output feature map **O** of shape $F \times H_O \times W_O$

1: for $(f = 0; f < \frac{F}{PF}; f++)$
2: for $(h = 0; h < H_O; h++)$
3: for $(w = 0; w < \frac{W_O}{PV}; w++)$
4: for $(c = 0; c < \frac{C}{PC}; c++)$
5: $\mathbf{O}[f][h][w] \mathrel{+}= \sum_{i=1}^{K-1} \sum_{j=1}^{K-1} \mathbf{W}[f][c][i][j] * \mathbf{I}[c][h*s+i][w*s+j]$

### *4.2. Standard Analytical Latency Model*

In this section, we outline the layer-wise processing latency model for the proposed accelerator, which constitutes the standard method as proposed in [14] for comparison.

The simplest form of a heuristic that estimates layer-wise latency on a hardware accelerator consists of partitioning the overall processing time to individual layers, $T_i$, corresponding to the time to perform one convolution in a feed-forward CNN consisting of $B$ convolutions/layers. The per-layer latency of an implemented CNN accelerator consists of three parts: (1) time for loading the input; (2) computation time; (3) time for storing the results.

The complete input has to be loaded into the on-chip memory only once for the first layer, while the partial results that do not fit into the on-chip memory are off-loaded to the off-chip memory. Nevertheless, the time spent on this memory transfer is assumed to be negligible.

The size of the weights and the input/output for convolution is shown in Table 2, following the notation defined in Table 1. The per-layer latency $T_i$ for a single convolutional layer $i; i = 1, \ldots, B$ of a CNN with $B$ layers is shown in Equations (3)–(5) as follows:

1. Loading time, i.e., the time to load the input into the on-chip memory. Note that the loading of the data is in parallel with respect to the channel parallelism $PC$:

$$\begin{aligned} T_{weights_i} &= \frac{K_i \times K_i \times F_i \times C_i \times DW}{PC \times PV \times M_{CLK} \times S \times M_{EFF}} \\ T_{data_i} &= \frac{H_{I_i} \times W_{I_i} \times C_i \times DW}{PC \times PV \times M_{CLK} \times S \times M_{EFF}} \\ T_{load_i} &= T_{weights_i} + T_{data_i} \end{aligned} \tag{3}$$

2. Computation time, i.e., the time to compute $PF \times PC$ parallel filters and channels, respectively:

$$T_{compute_i} = \frac{F_i \times C_i \times H_{I_i} \times W_{I_i} \times K_i \times K_i}{PF \times PC \times L_{CLK}} \tag{4}$$

3. Storing time, i.e., the time to store the output back to the off-chip memory. Note that similar to the input loading time, the storage time is divided by the channel parallelism $PC$:

$$T_{store_i} = \frac{H_{O_i} \times W_{O_i} \times F_i \times DW}{PC \times PV \times M_{CLK} \times S \times M_{EFF}} \tag{5}$$

Therefore, the time required to process a single convolutional layer can be written as in Equation (6) below:

$$T_i = \begin{cases} T_{i=1} & = T_{load_i} + T_{compute_i} \\ T_{i \neq 1 \vee N} & = max(T_{weights_i}, T_{compute_i}) \\ T_{i=N} & = max(T_{weights_i}, T_{compute_i}) + T_{store_i} \end{cases} \tag{6}$$

Note the *max* operations, which are present due to pipelining of the design, result in a latency determined by the slowest operation.

## 5. Experiments

In this section, we present the experimental settings, as well as the results with respect to both latency and energy estimation on different CNN architectures on the implemented accelerator (Section 4). The experiments were performed on an FPGA, as well as a custom ASIC. The networks were quantized into 8 bits [25], such that $DW = 8$ bits.

### *5.1. Evaluation for FPGA Design*

This section describes the accelerator on an Intel Arria GX 1150 FPGA, and we evaluate the proposed GP based method with respect to layer-wise latency estimation, while running CNNs on the accelerator. The fixed hardware parameters used for the FPGA implementation are such that the filter, channel and data parallelism were set as $PF = 64, PC = 64, PV = 1$. At the same time, the memory and logic clock frequencies were $M_{CLK} = 200$ MHz and $L_{CLK} = 200$ MHz. The memory efficiency was assumed to be $M_{EFF} = 70\%$, and the communicating data-width size was $S = 64$ bits. The evaluation dataset comprised of several different configurations of convolutional layers, which were the building blocks of three different CNNs, namely SSD [18] with 24 convolutions, Yolo [19] with 75 convolutions and ResNet-50 [17] with 57 convolutions. The characteristics of the dataset from a software perspective are shown in Table 3. These networks were chosen because their algorithmic structures present challenges to the accelerator design, its control and its scheduling. In particular, SSD and Yolo are characteristic by their irregularities, which result in the output being produced at different times, while ResNet is known for its residual blocks, which require implementing additional control in hardware.

**Table 3.** Dataset for the evaluation of the layer-wise latency on an FPGA.

| Parameter | Min | Mean | Max |
|---|---|---|---|
| $H_I/W_I$ | 1 | 42 | 418 |
| $H_O/W_O$ | 1 | 37 | 416 |
| $K$ | 1 | 2 | 7 |
| $C$ | 3 | 360 | 2048 |
| $F$ | 64 | 371 | 2048 |
| Latency (ms) | 0.018 | 0.841 | 11.727 |

In total, the dataset for layer-wise latency estimation for each layer $i$ consisted of $N = 156$ training samples, and the input feature size $M$ was 15, corresponding to: $H_{I_i}, W_{I_i}, H_{O_i}, W_{O_i}, K_i, F_i, C_i, PF, PC, PV, M_{CLK}, L_{CLK}, M_{EFF}, S$ and $DW$. The recorded latency per convolution represents the targets $\boldsymbol{y}$. Due to the limited size of the dataset, leave-one-out cross-validation (LOOCV) with respect to the mean absolute error (MAE) was used to compare the estimators. LOOCV is a particular case of leave-$k$-out cross-validation where $k = 1$, which means that a model is trained on all samples except one, on which the performance is then evaluated. Although potentially more expensive to implement, it provides a less biased estimate of the test errors. In this instance, the performance of the predictor is measured by the absolute error between the prediction and the target value. The error is accumulated for all samples from which the mean is then calculated by dividing the total summed error by the number of samples.

In the evaluation, the proposed method is compared with the standard analytical method, including LR, GTB and a fully-connected multi-layer NN. Due to the few data samples, we used the layer-wise latency model as presented in Section 4.2 as the mean function $m(.)$ of the GP model. We considered several hyperparameters for the proposed GP based method such as the learning rate, ranging from 0.1 to 0.000001 on a logarithmic scale, and the kernel, ranging from linear, Gaussian to Matérn kernels [23], and their combinations. The best parameters were found by a grid search with respect to the LOOCV MAE. For GTB and NN, we needed to determine the most influential parameters such as the learning rate, ranging from 0.01 to 0.0001 on a logarithmic scale, or for the GTB, the number of trees or the tree depth determined by gradual pruning. For the NN, we needed to decide the number of hidden nodes, between [10, 1], [10, 10, 1] and [10, 10, 10, 1], and for the activation function, we considered tanh, ReLU and sigmoid. The hyperparameters were similarly found through a grid search with respect to the LOOCV MAE. For the standard method and LR, it was not necessary to determine any hyperparameters. The results for latency estimation are presented in Table 4.

**Table 4.** Evaluation of layer-wise latency estimation for different methods on the convolutional neural network accelerator on an FPGA.

| Methods | Layer-Wise Latency LOOCV MAE (ms) | Implementation and Optimiser | Properties |
|---|---|---|---|
| Standard method | 0.450 | None | None |
| Linear regression | 0.450 | Sklearn [26] | Default |
| Gradient tree boosting | 0.607 | Sklearn [26]; AdaBoost [27] | Learning rate: 0.1<br>Number of trees: 10<br>Maximum depth: 3 |
| Neural network | 1.257 | TensorFlow [28]; Adam [29] | Batch size: 8<br>Learning rate: 0.1<br>Regulariser: L2, 0.001<br>Number of nodes: 10,10,1<br>Activations: ReLU |
| Our method | 0.312 | GPFlow [30]; Adam [29] | Mean function: $T_i$<br>Learning rate: 0.001<br>Kernel: Matérn 3/2 |

Overall, the best method proved to be the combination of the standard method as the mean function for the GP and the collected data. In comparison to other approaches, the proposed method achieved approximately a 30.7% improvement in LOOCV with respect to MAE, decreasing to 0.312 ms in comparison with the second best-performing methods, which were LR and the standard method with a 0.450 ms MAE.

### *5.2. Evaluation on the ASIC Design*

In this section, we implement the outlined hardware accelerator using 28 nm eASIC [31] technology on the Intel N3XS platform with 8GB DDR3 installed as an off-chip memory. The whole design was clocked at $M_{CLK}, L_{CLK} = 333$ MHz, and the $PF$, $PC$ and $PV$ were set as 64, 64 and 1, respectively. The example design we used in this experiment kept the same parallelism configuration for the entire CNN model. Other designs, such as the streaming design [32], can support layer-wise configurable parallelism. However, the layer-wise instantiation of a modern deep CNN requires extensive hardware resources, which are often not available.

Before the evaluation of our GP based estimation, we compare both the FPGA and eASIC implementations in terms of latency and power efficiency (frames per second per Watt (FPS/W)) on four CNN models including SSD, ResNet-50, Yolo and VGG-16. It can be clearly seen from Table 5 that the eASIC design achieved higher energy efficiency and smaller latency than the FPGA implementation on all four CNN models.

**Table 5.** Hardware performance comparison between the FPGA and eASIC design.

| | **SSD [18]** | | **ResNet-50 [17]** | | **Yolo [19]** | | **VGG-16 [33]** | |
|---|---|---|---|---|---|---|---|---|
| | Latency (ms) | FPS/W | Latency (ms) | FPS/W | Latency (ms) | FPS/W | Latency (ms) | FPS/W |
| FPGA | 3.24 | 7.01 | 4.62 | 4.92 | 41.22 | 0.55 | 23.18 | 0.98 |
| eASIC | 2.39 | 22.02 | 3.06 | 17.20 | 31.55 | 1.67 | 15.35 | 3.43 |

Next, we evaluated the GP based estimation for the eASIC design with respect to latency and energy consumption. Instead of estimating per-layer latency, this experiment aimed at validating the GP based estimation of a whole NN for both latency and energy consumption. We ran ResNet-50 [17] using different network configurations with respect to energy and latency to form the evaluation and training datasets, which is illustrated in Figure 3.

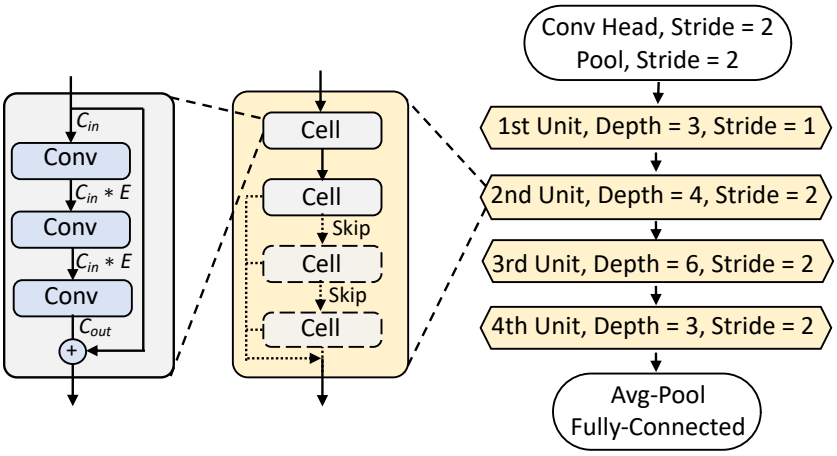


**Figure 3.** ResNet-50 with different depths, channel numbers and expansion ratios.

The network contains three parts: head part, middle part and tail part. The head part includes a convolutional layer and a pooling layer with stride-2, while the tail part consists of an average pooling layer followed by a fully-connected layer. We fixed the head and tail parts while changing the network configurations for the middle part that contains four residual blocks with a gradually reduced feature map size and increased channel numbers. In each residual block, the depth ranges from two to $D_i$, where $D_i$ denotes the maximal depth in the $i$th block. In each cell of the residual block, the expansion ratio ($E$) was chosen from [0.5, 0.75, 1.0]. For regression, as the hardware properties are fixed for the eASIC design, we only needed to encode the network configurations as a 13-dimensional vector, which represents the expansion ratio used in the 13 cells, giving $M = 13$. The expansion ratio was zero, if this cell was skipped. We randomly sampled 800 different network configurations and evaluated these networks on our eASIC designs with respect to latency and energy consumption. We used 600 samples for training and 200 samples for evaluation. Therefore, even though the hardware configuration remained fixed, we benchmarked the methodology with respect to changing various software parameters.

To demonstrate the advantages of GP based estimation compared with other regression techniques, we also compared it with LR, GTB and NN, which is illustrated in Table 6. In this instance we used a zero mean function, such that the methods should rely more on data, instead of any bias that could have been potentially induced by inaccurate analytical approximation. All methods used the same hyperparameters as in Section 5.1, to demonstrate the flexibility and simplicity of the implementation of the proposed GP regression model. It can be seen that our method achieved a smaller MAE on both latency and energy estimation, when compared with the other methods. In comparison to LR, which is a simple and widely adopted estimator, the performance can be improved by approximately two times with respect to both latency and energy estimates.

**Table 6.** Evaluation of network-wise latency and energy estimation for different methods on the convolutional neural network accelerator on an eASIC.

| Methods | Latency MAE (ms) | Energy MAE (W) | Implementation and Optimiser | Properties |
|---|---|---|---|---|
| Linear regression | 0.177 | 0.272 | Sklearn [26] | Default |
| Gradient tree boosting | 0.476 | 0.501 | Sklearn [26]; AdaBoost [27] | Learning rate: 0.1<br>Number of trees: 10<br>Maximum depth: 3 |
| Neural network | 0.108 | 0.241 | TensorFlow [28]; Adam [29] | Batch size: 8<br>Learning rate: 0.1<br>Regulariser: L2, 0.001<br>Number of nodes: 10,10,1<br>Activations: ReLU |
| Our method | 0.079 | 0.151 | GPFlow [30]; Adam [29] | Mean function: 0<br>Learning rate: 0.001<br>Kernel: Matérn 3/2 |

Furthermore, in Figure 4, we show the advantages of GP over the aforementioned methods on smaller datasets by varying the training dataset size and number of features as the input of the models with respect to the overall prediction latency and energy consumption on the eASIC. Each experiment was repeated three times varying the number of available data points or features to evaluate the robustness of the compared methods. It can be observed that the GP is more accurate and also more robust as the standard deviation is consistently smaller in comparison to the other methods in all experiments.

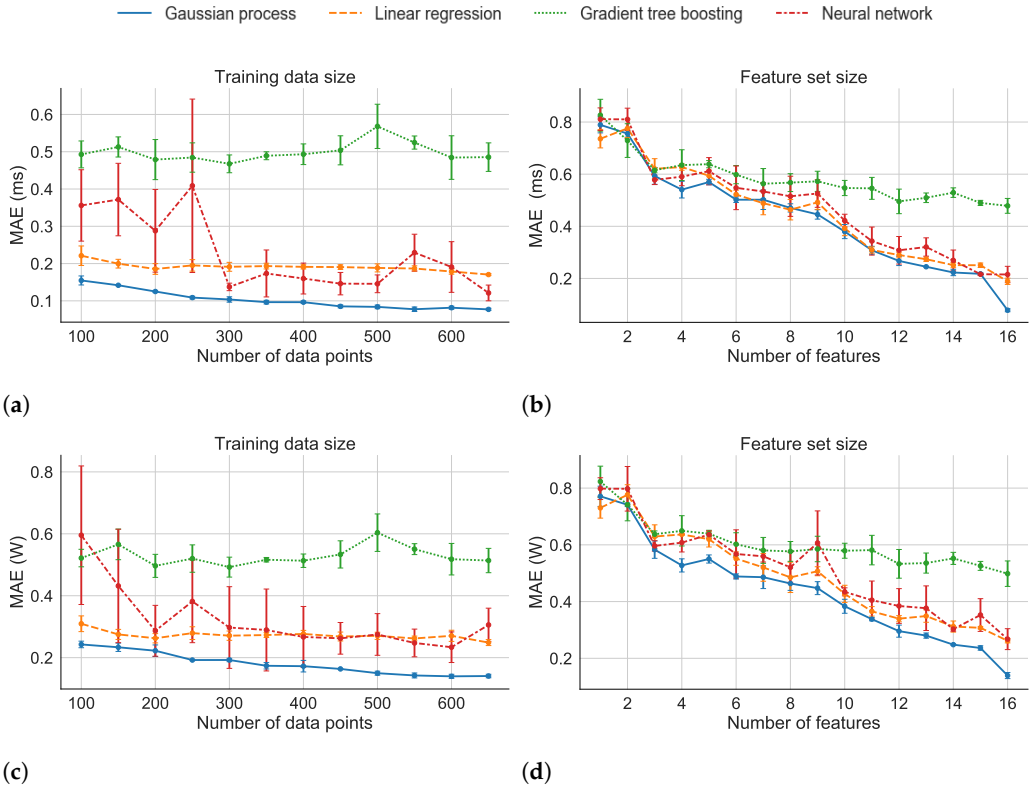


**Figure 4.** Prediction benchmarks for latency with respect to changing training data size (**a**) and feature set size (**b**). Benchmarks for energy with respect to changing training data size (**c**) and feature set size (**d**).

The main advantage of the proposed method lays in its implementation simplicity, as it reuses those variables that can be commonly found in hardware or algorithmic datasheets and commonly used in DSE, combined with recorded measurements. The method can be improved by recording more measurements and simple fine-tuning of the hyperparameters related to the kernel $\boldsymbol{K}$. Nevertheless, as demonstrated in Sections 5.1 and 5.2, the method is capable of estimating the performance even with respect to few collected data samples.

A potential limitation of this method, as was eluded to in Section 3, stems from the kernel computation, which scales with the complexity of $\mathcal{O}(N^3)$. This means that the inference time can be prolonged if there are many training samples. One possible solution to overcome this problem is to use variational inference to determine the $k$ most important points that have to be included in the kernel computation [34]. Nevertheless, the inference time is much less than the time needed for synthesis and then running the design on hardware.

## 6. Conclusions

In this paper, we propose an accurate method for estimating the performance of an accelerator for convolutional neural networks and compare it with the standard method, linear regression, gradient tree boosting and an artificial neural network. Moreover, we evaluate our method with respect to two hardware platforms on which we accurately predict the overall latency or energy consumption of the given convolutional neural networks. The evaluation demonstrates that the innovative Gaussian process method paired with collected data can provide an accuracy improvement with respect to the other compared methods. Future work includes providing tools to automate our approach, and extending it to cover applications beyond machine learning designs.

**Author Contributions:** Conceptualization, M.F., H.F. and D.M.; data curation, M.F. and H.F.; investigation, M.F. and H.F.; resources, S.L. and X.N.; supervision, W.L.; validation, M.F. and H.F.;

writing—original draft, M.F., H.F. and D.M.; writing—review and editing, M.F., H.F., D.M., H.Z., S.L. and X.N. All authors read and agreed to the published version of the manuscript.

**Funding:** The support of the U.K. EPSRC (EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/S030069/1), Corerain, Intel and Xilinx is gratefully acknowledged.

**Data Availability Statement:** The data presented in this study are available on request from the corresponding author.

**Acknowledgments:** We thank Yann Herklotz, Alexander Montgomerie-Corcoran, the ARC'20 and Electronics reviewers for insightful suggestions.

**Conflicts of Interest:** The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; nor in the decision to publish the results.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| ASIC | Application-specific integrated circuit |
| CPU | Central processing unit |
| CNN | Convolutional neural network |
| DSE | Design space exploration |
| FPGA | Field-programmable gate array |
| GP | Gaussian process |
| GPU | Graphical processing unit |
| GTB | Gradient tree boosting |
| LOOCV | Leave-one-out cross-validation |
| LR | Linear regression |
| MAE | Mean absolute error |
| NN | Neural network |

## References

1. Ferianc, M.; Fan, H.; Rodrigues, M. VINNAS: Variational Inference based Neural Network Architecture Search. *arXiv* **2020**, arXiv:2007.06103.
2. Fan, H.; Liu, S.; Ferianc, M.; Ng, H.C.; Que, Z.; Liu, S.; Niu, X.; Luk, W. A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA. In Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT), Naha Okinawa, Japan, 11–15 December 2018; pp. 14–21.
3. Liu, S.; Luk, W. Towards an Efficient Accelerator for DNN-Based Remote Sensing Image Segmentation on FPGAs. In Proceedings of the 2019 29th International Conference on Field Programmable Logic and Applications (FPL), Barcelona, Spain, 8–12 September 2019; pp. 187–193.
4. Brown, T.B.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. Language models are few-shot learners. *arXiv* **2020**, arXiv:2005.14165.
5. Kwon, Y.; Won, J.H.; Kim, B.J.; Paik, M.C. Uncertainty quantification using Bayesian neural networks in classification: Application to biomedical image segmentation. *Comput. Stat. Data Anal.* **2020**, *142*, 106816. [CrossRef]
6. McAllister, R.; Gal, Y.; Kendall, A.; Van Der Wilk, M.; Shah, A.; Cipolla, R.; Weller, A. Concrete Problems for Autonomous Vehicle Safety: Advantages of Bayesian Deep Learning. In Proceedings of the 26th International Joint Conference on Artificial Intelligence (IJCAI'17), Buenos Aires, Argentina, 25–31 July 2017; pp. 4745–4753.
7. Xuan-Mung, N.; Hong, S.K. Barometric altitude measurement fault diagnosis for the improvement of quadcopter altitude control. In Proceedings of the 2019 19th International Conference on Control, Automation and Systems (ICCAS), Jeju, Korea, 15–18 October 2019; pp. 1359–1364.
8. Park, D.; Yu, H.; Xuan-Mung, N.; Lee, J.; Hong, S.K. Multicopter PID Attitude Controller Gain Auto-tuning through Reinforcement Learning Neural Networks. In Proceedings of the 2019 2nd International Conference on Control and Robot Technology, Phuket, Thailand, 25–27 October 2019; pp. 80–84.
9. Nguyen, N.P.; Mung, N.X.; Thanh Ha, L.N.N.; Huynh, T.T.; Hong, S.K. Finite-Time Attitude Fault Tolerant Control of Quadcopter System via Neural Networks. *Mathematics* **2020**, *8*, 1541. [CrossRef]
10. Mittal, S. A survey of FPGA based accelerators for convolutional neural networks. *Neural Comput. Appl.* **2020**, *32*, 1109–1139. [CrossRef]

11. Rahman, A.; Oh, S.; Lee, J.; Choi, K. Design space exploration of FPGA accelerators for convolutional neural networks. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1147–1152.
12. Yasudo, R.; Coutinho, J.; Varbanescu, A.; Luk, W.; Amano, H.; Becker, T. Performance Estimation for Exascale Reconfigurable Dataflow Platforms. In Proceedings of the 2018 International Conference on Field-Programmable Technology (FPT), Naha Okinawa, Japan, 11–15 December 2018; pp. 314–317.
13. Dai, S.; Zhou, Y.; Zhang, H.; Ustun, E.; Young, E.F.; Zhang, Z. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In Proceedings of the 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 29 April–1 May 2018; pp. 129–132.
14. Liu, S.; Fan, H.; Niu, X.; Ng, H.C.; Chu, Y.; Luk, W. Optimizing CNN based Segmentation with Deeply Customized Convolutional and Deconvolutional Architectures on FPGA. *ACM Trans. Reconfig. Technol. Syst.* **2018**, *11*, 1–22. [CrossRef]
15. Williams, C.K.; Rasmussen, C.E. Gaussian processes for regression. *Adv. Neural Inf. Process. Syst.* **1996**, *8*, 514–520.
16. Ferianc, M.; Fan, H.; Chu, R.S.; Stano, J.; Luk, W. Improving Performance Estimation for FPGA-Based Accelerators for Convolutional Neural Networks. In *International Symposium on Applied Reconfigurable Computing*; Springer: Berlin, Germany, 2020; pp. 3–13.
17. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; Volume 2016, pp. 770–778.
18. Liu, W.; Anguelov, D.; Erhan, D.; Szegedy, C.; Reed, S.; Fu, C.Y.; Berg, A. SSD: Single shot multibox detector. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*; Springer: Berlin, Germany, 2016; Volume 9905, pp. 21–37.
19. Redmon, J.; Divvala, S.; Girshick, R.; Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. In Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 27–30 June 2016; Volume 3, pp. 779–788.
20. LeCun, Y.; Boser, B.; Denker, J.S.; Henderson, D.; Howard, R.E.; Hubbard, W.; Jackel, L.D. Backpropagation applied to handwritten zip code recognition. *Neural Comput.* **1989**, *1*, 541–551. [CrossRef]
21. Fan, H.; Luo, C.; Zeng, C.; Ferianc, M.; Que, Z.; Liu, S.; Niu, X.; Luk, W. F-E3D: FPGA based Acceleration of an Efficient 3D Convolutional Neural Network for Human Action Recognition. In Proceedings of the 2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP), New York, NY, USA, 15–17 July 2019; Volume 2160, pp. 1–8.
22. Venieris, S.; Kouris, A.; Bouganis, C.S. *Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions*; ACM: New York, NY, USA, 2018; Volume 51, pp. 1–39.
23. Rasmussen, C.E. *Gaussian Processes in Machine Learning*; The MIT Press: Cambridge, MA, USA, 2005.
24. Ioffe, S.; Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv* **2015**, arXiv:1502.03167.
25. Jacob, B.; Kligys, S.; Chen, B.; Zhu, M.; Tang, M.; Howard, A.; Adam, H.; Kalenichenko, D. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Salt Lake City, UT, USA, 18–23 June 2018; pp. 2704–2713.
26. Pedregosa, F.; Varoquaux, G.; Gramfort, A.; Michel, V.; Thirion, B.; Grisel, O.; Blondel, M.; Prettenhofer, P.; Weiss, R.; Dubourg, V.; et al. Scikit-learn: Machine Learning in Python. *J. Mach. Learn. Res.* **2011**, *12*, 2825–2830.
27. Friedman, J.H. Stochastic gradient boosting. In *Computational Statistics & Data Analysis*; Elsevier: Amsterdam, The Netherlands, 2002; Volume 38, pp. 367–378.
28. Abadi, M.; Agarwal, A.; Barham, P.; Brevdo, E.; Chen, Z.; Citro, C.; Davis, A.; Dean, J.; Devin, M.; Ghemawat, S.; et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015. Available online: https://www.tensorflow.org/ (accessed on 14 December 2020).
29. Kingma, D.P.; Ba, J. Adam: A method for stochastic optimization. *arXiv* **2014**, arXiv:1412.6980.
30. Matthews, D.G.; Alexander, G.; Van Der Wilk, M.; Nickson, T.; Fujii, K.; Boukouvalas, A.; León-Villagrá, P.; Ghahramani, Z.; Hensman, J. GPflow: A Gaussian process library using TensorFlow. *J. Mach. Learn. Res.* **2017**, *18*, 1299–1304.
31. Intel Corporation. eASIC Technology. 2018. Available online: https://www.intel.co.uk/content/www/uk/en/products/programmable/asic/easic-devices.html (accessed on 2 December 2020).
32. Venieris, S.I.; Bouganis, C.S. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In Proceedings of the 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Washington, DC, USA, 1–3 May 2016; pp. 40–47.
33. Simonyan, K.; Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv* **2014**, arXiv:1409.1556.
34. Titsias, M. Variational learning of inducing variables in sparse Gaussian processes. In Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics (AISTATS), Clearwater Beach, FL, USA, 16–18 April 2009; pp. 567–574.