X-TrainCaps: Accelerated Training of Capsule Nets through Lightweight Software Optimizations

Alberto Marchisio[1], Beatrice Bussolino[1,2], Alessio Colucci[2], Muhammad Abdullah Hanif[1], Maurizio Martina[2], Guido Masera[2] and Muhammad Shafique[1]

[1]Technische Universität Wien (TU Wien), Vienna, Austria
[2]Politecnico di Torino (PoliTo), Turin, Italy
Email: {alberto.marchisio, muhammad.hanif, muhammad.shafique}@tuwien.ac.at
{beatrice.bussolino, alessio.colucci}@studenti.polito.it
{maurizio.martina, guido.masera}@polito.it

Abstract

Convolutional Neural Networks (CNNs) are extensively in use due to their excellent results in various machine learning (ML) tasks like image classification and object detection. Recently, Capsule Networks (CapsNets) have shown improved performances compared to the traditional CNNs, by encoding and preserving spatial relationships between the detected features in a better way. This is achieved through the so-called Capsules (i.e., groups of neurons) that encode both the instantiation probability and the spatial information. However, one of the major hurdles in the wide adoption of CapsNets is its gigantic training time, which is primarily due to the relatively higher complexity of its constituting elements. In this paper, we illustrate how can we devise new optimizations in the training process to achieve fast training of CapsNets, and if such optimizations affect the network accuracy or not. Towards this, we propose a novel framework "X-TrainCaps" that employs lightweight software-level optimizations, including a novel learning rate policy called *WarmAdaBatch* that jointly performs *warm restarts* and *adaptive batch size*, as well as *weight sharing* for capsule layers to reduce the hardware requirements of CapsNets by removing unused/redundant connections and capsules, while keeping high accuracy through tests of different learning rate policies and batch sizes. We demonstrate that one of the solutions generated by X-TrainCaps framework can achieve 58.6% training time reduction while preserving the accuracy (even 0.9% accuracy improvement), compared to the CapsNet in the original paper by Sabour et al. (2017), while other Pareto-optimal solutions can be leveraged to realize trade-offs between training time and achieved accuracy.

1 Introduction

Recently the development of Deep Neural Networks (DNNs), especially Convolutional Neural Networks (CNNs), has seen a dramatic increase, leading to many different architectures. An important computing challenge is the optimization of CNNs and their hyper-parameters. Most of the fine-tuning optimizations (e.g., choosing the training policy, the learning rate, the optimizer, etc.) are repetitive and time-consuming, because every change must be tested with many epochs of training and repeated many times to have statistical significance. This is significantly worsened by increasing the CNN complexity, which leads to a more demanding compute effort to find the right set of optimizations.

Different methods have been explored in the literature to reduce the training time of CNNs, such as one-cycle policy by Smith (2018), *warm restarts* by Loshchilov and Hutter (2016), *adaptive batch size* by De et al. (2016), Goyal et al. (2017), Devarakonda et al. (2017). However, a key limitation of CNNs is: no information on the spatial correlation between detected features is retained. This causes poor network performances in terms of accuracy when the object to be recognized is rotated, has a different orientation, or presents any other geometrical variation. Currently, this problem is solved by training networks on expanded datasets, that include also transformed and modified objects. However, wider datasets lead to much longer training times. Longer training times not only pose

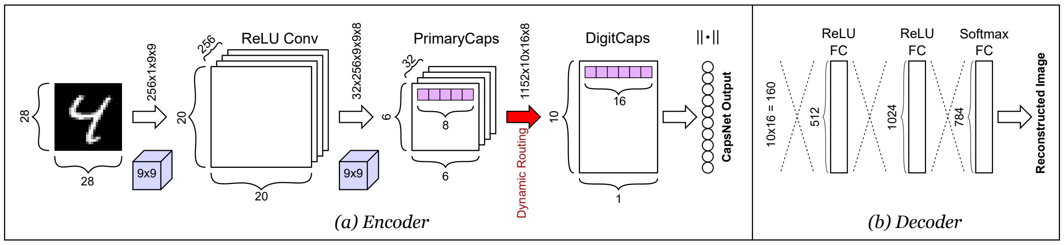


Figure 1: Architectural view of a CapsNet.

delays in the DNN development cycle, but also make it a complex job for a wide range of developers, requiring super-costly training machines like Nvidia's DGX-2 or rely on outsourcing to third party cloud services that can compromise privacy and security requirements. Hence, both advanced DNN architectures and fast training techniques are necessary.

Capsule Networks (CapsNets, see Figure 1) aim to solve the limitation of CNNs (w.r.t. preserving the spatial correlation between detection features) by substituting single neurons with so called Capsules (i.e., groups of neurons) and routing by agreement between capsule layers, which provide the ability to encode both the instantiation probability of an object and its instantiation parameters (width, orientation, skew, etc.). However, one of the major hurdles in the adoption of CapsNets is their gigantic training time, which is primarily due to the higher complexity of their constituting elements.

To address this challenge, we present X-TrainCaps, a systematic methodology to employ different optimization methods for significantly reducing the training time and number of parameters of CapsNets, while preserving or improving their accuracy[1]. However, this requires a study of impact of different optimizations (like learning rate policy, batch size adaptation, etc.) on the training time and accuracy of CapsNets. X-TrainCaps provides multiple Pareto-optimal solutions that can be leveraged to trade-off training time reductions with no / tolerable accuracy loss. For instance, in our experimental evaluations, X-TrainCaps provides a solution that can reduce the training time of CapsNets by 58.6% while providing a slight increase in its accuracy (i.e. 0.9%). Another solution provides 15% reduction in the number of parameters of the CapsNet without affecting its accuracy.

Our Novel Contributions:

- Analysis of the CapsNets behavior when different learning rate policies (like *one-cycle policy* or *warm restarts*) are applied (Section 3).
- A novel training methodology, X-TrainCaps, which accelerates the training of CapsNets by combining *warm restarts*, *adaptive batch size* and *weight sharing* in a automated flow specialized for the nature of CapsNets like capsule and different capsule layers (Section 4).
- Reduction of redundant parameters via *weight sharing* and a smaller decoder spaecialized for CapsNets. This optimizations reduce the number of parameters by more than 15% (Sections 4.3 & 4.4).

Before proceeding to the technical sections, we present an overview of CapsNets and the learning rate policies in Section 2, to a level of detail necessary to understand the contributions of this paper.

2 Background and Related Work

2.1 CapsNets

Capsules were introduced by Hinton, Krizhevsky, et al. (2011). A capsule is a group of neurons that encodes both the instantiation probability of an object and the spatial information. Sabour et al. (2017) and Hinton, Sabour, et al. (2018) introduced two architectures based on capsules, tested on MNIST (LeCun et al. (1998)), with better accuracy compared to the state-of-the art traditional CNNs. The architecture of the CapsNet that we use in our work corresponds to the model of Sabour et al.

[1]Our methodology may also be beneficial for other complex CNNs, as it enables integration of available open-source optimizations in the DNN training loop.

(2017), as shown in Figure 1. The layers composing the encoder are: an initial convolutional layer (ConvLayer), a PrimaryCaps layer, which transforms the scalars numbers of ConvLayer in vectors, and a DigitCaps layer, which outputs the probabilities of the digits being present in the input image. The encoding network is followed by a decoder, composed by three fully-connected layers, which output the reconstructed image. The loss computed on the reconstructed image (Reconstruction Loss) encourages the capsules of the DigitCaps layer to encode the instantiation parameters of the object.

2.2 Learning Rate Schedules

One-Cycle Policy: This method, proposed by Smith (2015) and Smith (2018), consists of three phases of training. In the first phase, the leaning rate is linearly increased from a minimum to a maximum value in an optimal range. In the second phase, the learning rate is symmetrically decreased. In a small fraction of the last steps, the learning rate must be annealed to a very low value. In Equation 1 the formulas of the three parts of one cycle policy are reported: ts is the training step, TS is the total number of steps in the training epochs, lr_{min} and lr_{max} are the learning rate range boundaries. Saddle points in the loss function slow down the training process, since the gradients in these regions have smaller values. Increasing the learning rate helps to faster traverse the saddle points.

$$\begin{cases} lr = lr_{min} + ts \cdot \frac{lr_{max} - lr_{min}}{0.45 \cdot TS} & 0 < ts < 0.45 \cdot TS \\ lr = lr_{min} + (ts - 0.9 \cdot TS) \cdot \frac{lr_{min} - lr_{max}}{0.45 \cdot TS} & 0.45TS < ts < 0.9TS \\ lr = lr_{min} - 9 \cdot \frac{lr_{min}}{TS} \cdot (ts - 0.9 \cdot TS) & 0.9 \cdot TS < ts < TS \end{cases} \tag{1}$$

Warm Restarts: Stochastic Gradient Descent with Warm Restarts technique was proposed by Loshchilov and Hutter (2016). The learning rate is initialized to a maximum value and then it is decreased with cosine annealing until reaching the lower bound of a chosen interval. When the learning rate reaches the minimum value, it is set again to the maximum value, realizing a step function. The cosine annealing function which is used is reported in Equation 2: lr_{min} and lr_{max} are learning rate range boundaries, ts is the training step, T_i is the number of training steps for each cycle. When $ts = Ti$, ts is set to 0 and the cycle starts again. This process is repeated cyclically during the whole training time. The period of a cycle needs to be properly set to optimize the training time and the accuracy. Increasing step-wise the learning rate emulates a warm restart of the network and encourages the model to step out from possible local minima or saddle points.

$$lr = lr_{min} + \frac{1}{2} (lr_{max} - lr_{min}) \left(1 + cos\left(\pi \cdot \frac{ts}{T_i}\right)\right) \tag{2}$$

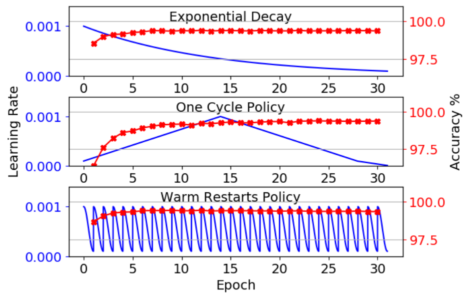
2.3 Adaptive Batch Size

When training a DNN, a small batch size can provide a faster convergence, while a larger batch size allows to have an higher data parallelism and, consequently, high computational efficiency. For this reason, many authors have studied methods to increase the batch size with fixed schedules (Babanezhad et al. (2015), Daneshmand et al. (2016)) or following an adaptive criterion (De et al. (2016), Goyal et al. (2017), Devarakonda et al. (2017)).

3 Analysis of Learning Rate Schedules applied to CapsNets

The techniques described in Section 2 have been tailored for traditional CNNs, to improve their performances in terms of accuracy and training time. This section aims to study whether different learning rate policies and batch size selection are effective when applied for training the CapsNets. Since the traditional neurons of the CNNs are replaced by multidimensional capsules in the CapsNets, the number of parameters (weights and biases) to be trained is huge.

For this purpose, we implemented different state-of-the-art learning rate policies for the training loop of the CapsNet, such that *these techniques are enhanced for the capsule structures and relevant parameters of the CapsNet*. Figure 2 (a) shows the learning rate changes for different techniques and how the accuracy of the CapsNet varies accordingly. More detailed results of our analyses, including the comparisons with the LeNet5, are reported in the Table of Figure 2 (b).



(a) Evaluating Different Learning Rate Policies on the CapsNet: Learning Rate vs. Accuracy for Different Epochs when using Different Learning Rate Policies

	LeNet5			CapsNet		
	Max. Accuracy	Epoch of max Accuracy	Epochs to reach acc. of fixed l.r.	Max. Accuracy	Epoch of max Accuracy	Epochs to reach acc. of fixed l.r.
Fixed	98.860	17	17	99.366	29	29
Exp. Decay	99.238	28	6	99.404	12	7
One Cycle P.	99.218	30	19	99.384	24	23
Warm Restarts	99.234	20	4	99.440	11	6
AdaBatch	99.178	19	5	99.410	8	5

(b) Comparison between LeNet5 and CapsNet for Different Learning Rate Policies for Training the Networks

Figure 2: **(a)** In blue, how learning rate changes when different learning rate policies like exponential decay, one cycle policy and *warm restarts* are applied. In red, the accuracy reached by CapsNet at every epoch of training with the corresponding learning rate policy applied. **(b)** A table summarizing the comparative differences between LeNet5 and CapsNet when the same learning rate policies are applied. For each architecture, different columns of the table show: the maximum reached accuracy, training epochs[2]to reach the maximum accuracy and the training epochs to reach the same accuracy of the network when fixed learning rate policy is used.

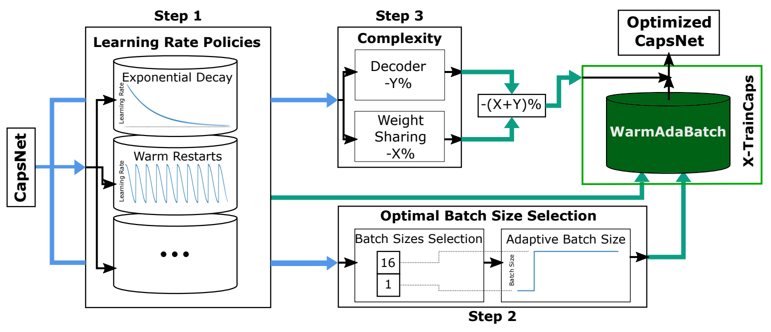


Figure 3: X-TrainCaps flow: the CapsNet at the input goes through the different stages of optimization in parallel, to search for the right learning rate policy, batch size and complexity reduction, obtaining at the output the Optimized CapsNet, based on the optimization criteria chosen by the user.

From the results of this analisys, we can derive the following **key observations**:

1. The *warm restarts* technique is the most promising because it allows to reach the same accuracy (99.366%) as the CapsNet with a fixed learning rate, providing a reduction of 79.31% in the training time.
2. A more extensive training with *warm restarts* leads to to an accuracy improvement of 0.074%.
3. The *adaptive batch size* shows similar improvements in terms of accuracy (99.41%) and number of training epochs.
4. The first epochs with smaller batch size execute in a longer time.

4 X-TrainCaps: Our Framework to Accelerate the Training of CapsNets

Training a CapsNet consists of a multi-objective optimization problem, because our scope is to maximize the accuracy, while minimizing the training time and the network complexity. The processing flow of our X-TrainCaps framework is shown in Figure 3. Before describing how to integrate different optimizations in an automated training methodology and how to generate the optimized CapsNet at the output (Section 4.5), we present how these optimizations have been

[2]Training times for a single epoch are different between LeNet5 and CapsNet: the former takes an average of 17 seconds using the fixed learning rate, while the latter takes 295 seconds.

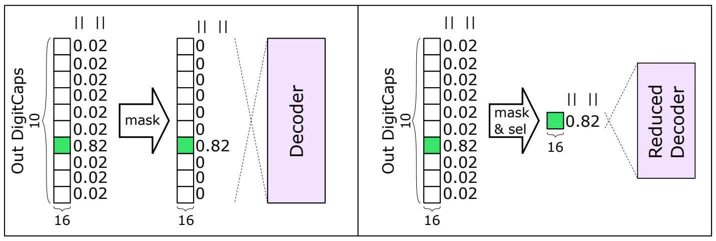


Figure 4: (left) All the OutDigitCaps outputs apart the one with highest magnitude are set to zero. Then the decoder receives 10x16 inputs. **(right)** Only the OutDigitCaps output with highest magnitude is fed to a reduced decoder, with 1x16 inputs.

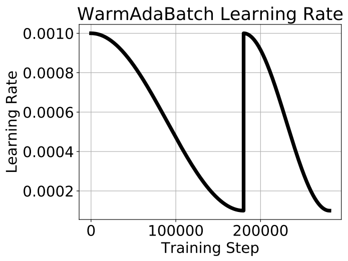


Figure 5: Learning rate policy for our *WarmAdaBatch*.

implemented with enhancements for the CapsNets, which is necessary to realize an integrated training methodology.

4.1 Learning Rate Policies for CapsNets

The first parameter analyzed to improve the training process of CapsNets is the learning rate. The optimal learning rate range is evaluated with the range boundaries 0.0001 and 0.001. For our framework, we use the following parameters in these learning rate policies:

- **Fixed learning rate**: 0.001;
- **Exponential decay**: starting value 0.001, decay rate 0.96, decay steps 2000;

$$lr = lr_0 \cdot 0.96^{current\_step/2000}$$

- **One cycle policy**: lower bound 0.0001, upper bound 0.001, annealing to 10^{-5} in the last 10% of training steps (see Algorithm 1);
- **Warm restarts**: lower bound 0.0001, upper bound 0.001, cycle length = one epoch (see Algorithm 2).

4.2 Batch Size

To realize *adaptive batch size*, the batch size is set to 1 for the first three epochs, and then increased every five epochs for three times. In particular, the user can choose a value P and batch size will assume values 2^P, 2^{P+1} and 2^{P+2} (see Algorithm 3).

4.3 Complexity of Decoder

The decoder is an essential component of the CapsNet. Indeed, absence of a decoder results in a lower accuracy of the CapsNet. The outputs of DigitCaps are fed to the decoder: the output of the capsule (a vector) with the highest value is left untouched, while the remaining 9 vectors are set to zero (Figure 4 left). Thus, the decoder receives 10×16 values, where 9×16 are null. Therefore, we optimize the model by using a reduced decoder (Figure 4 right) with only the 1×16 inputs which are linked to the capsule that outputs the highest probability. Overall, the original decoder has 1.4M parameters (weights and biases) while the reduced decoder provides a 5% reduction, with 1.3M parameters.

4.4 Complexity Reduction through Weight Sharing

Algorithm 4 illustrates how to share the weights between the PrimaryCaps and the DigitCaps layers, by having a single tensor weight in common for all the 8-element vectors inside each 6x6 capsule. Using this method, it is possible to reduce the total number of parameters by more than 15%, from 8.2 millions to 6.7 millions. However, the accuracy drops by almost 0.3%, when comparing it to the baseline CapsNet.

4.5 WarmAdaBatch

Among the explored learning rate policies, *warm restarts* guarantees the most promising in terms of accuracy, while *adaptive batch size* provides a good trade-off to obtain fast convergence. We propose *WarmAdaBatch* (see Algorithm 5) learning rate policy to expand the space of the solutions by combining the best of two worlds. For the first three epochs, the batch size is set to 1, then it is increased to 16 for the remaining training time. A first cycle of *warm restarts* policy is done during the first three epochs, a second one during the remaining training epochs. The learning rate variation of the *WarmAdaBatch* is shown in Figure 5.

4.6 Optimization Choices

Our framework is able to automatically optimize CapsNets and its training depending on which parameters the user wants to improve: using *WarmAdaBatch*, the accuracy and the training time are automatically co-optimized. The number of parameters can be reduced, at the cost of some accuracy loss and training time increase, by enabling the *weight sharing*, along with *WarmAdaBatch*.

Algorithm 1 One Cycle Policy for CapsNet

```
 1: procedure OneCyclePolicy($lr_{min}, lr_{max}, TotalSteps, Tcurr$)
 2:     $t_m \leftarrow 0.45 TotalSteps$
 3:     $m \leftarrow \frac{lr_{max}-lr_{min}}{t_m}$
 4:     $m_{ann} \leftarrow 9\frac{lr_{min}}{TotalSteps}$
 5:     if $Tcurr \leq t_m$ then
 6:         $lr \leftarrow mx + lr_{min}$
 7:     else if $t_m \leq Tcurr \leq 2t_m$ then
 8:         $lr \leftarrow -m(x-2t_m) + lr_{min}$
 9:     else
10:         $lr \leftarrow -m_{ann}(x-2t_m) + lr_{min}$
11:     end if
12: end procedure
```

Algorithm 2 Warm Restarts for CapsNet: learning rate is decayed with cosine annealing.

```
1: procedure WarmRestarts($lr_{min}, lr_{max}, T_{curr}, T_i$)
2:     $lr \leftarrow lr_{min} + \frac{1}{2}(lr_{max} - lr_{min})\left(1 + \cos \pi \frac{T_{curr}}{T_i}\right)$        ▷ Learning rate update
3:     if $T_{curr} = T_i$ then
4:         $T_{curr} \leftarrow 0$        ▷ Warm Restart after $T_i$ training steps
5:     else
6:         $T_{curr} \leftarrow T_{curr} + 1$        ▷ Current step update
7:     end if
8:     return $T_{curr}$
9: end procedure
```

Algorithm 3 AdaBatch for CapsNet: Batch Size is increased during training.

```
 1: procedure AdaBatch($P, CurrentEpoch$)
 2:     if $CurrentEpoch \leq 3$ then
 3:         $BatchSize \leftarrow 1$
 4:     else if $4 \leq CurrentEpoch \leq 8$ then
 5:         $BatchSize \leftarrow 2^P$
 6:     else if $9 \leq CurrentEpoch \leq 13$ then
 7:         $BatchSize \leftarrow 2^{P+1}$
 8:     else
 9:         $BatchSize \leftarrow 2^{P+2}$
10:     end if
11: end procedure
```

Algorithm 4 Weight Sharing for CapsNet, it is applied only in the DigitCaps layer

```
 1:                                      ▷ BatchSize represents the dimension containing the single elements
 2: procedure DIGITCAPS($input, BatchSize$)                        ▷ Input size is [BatchSize, 32, 36, 8]
 3:     $initialize\ weight$                                 ▷ Weight size is [BatchSize, 32, 1, 10, 16, 8]
 4:     $initialize\ bias$                                         ▷ Bias size is [BatchSize, 1, 10, 16, 1]
 5:     for $Step = 1, 36$ do                              ▷ We move along the dimension with 36 elements
 6:                                                           ▷ Result size is [bs, 32, 36, 10, 16, 1]
 7:         $u[Step] \leftarrow matrix\_multiply(weight[1], input[Step])$
 8:                                         ▷ We use always the same weight, instead of cycling
 9:     end for
10:     $v \leftarrow routing(u, bias)$                           ▷ Output size is [BatchSize, 1, 10, 16, 1]
11:     return $v$
12: end procedure
```

Algorithm 5 Our WarmAdaBatch for CapsNet

```
 1: procedure WARMADABATCH($lr_{min}$, $lr_{max}$, $MaxEpoch$, $MaxStep$)
 2:     $T_{curr} \leftarrow 0$
 3:     for $Epoch = 1, MaxEpoch$ do
 4:         $Adabatch(4, Epoch)$                                                   ▷ Batch size update
 5:         if $Epoch \leq 3$ then
 6:             $T_i \leftarrow 3 * 60000$                          ▷ Steps in 3 epochs with batch size 1
 7:         else
 8:             $T_i \leftarrow 27 * 3750$                        ▷ Steps in 27 epochs with batch size 16
 9:         end if
10:         for $Step = 1, MaxStep$ do
11:             $T_{curr} \leftarrow WarmRestarts(lr_{min}, lr_{max}, T_{curr}, T_i)$        ▷ Learning Rate update
12:         end for
13:     end for
14: end procedure
```

5 Evaluation

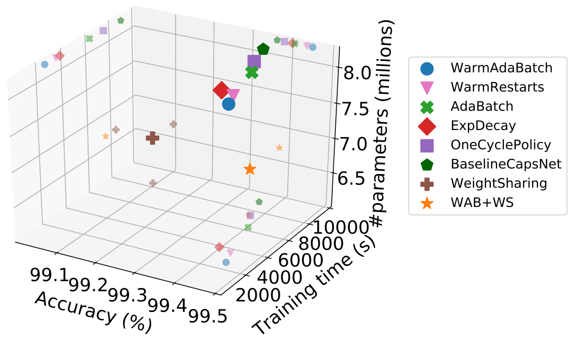
5.1 Experimental Setup

We developed our framework using PyTorch library (Paszke et al. (2017)), running on two Nvidia GTX 1080 Ti GPUs. We tested it on the MNIST dataset (LeCun et al. (1998)), since the original CapsNet proposed by Sabour et al. (2017) is tailored on it. The dataset is composed of 60000 samples for training and 10000 test samples. After each training epoch, a test is performed. At the beginning of each epoch, the samples for training are randomly shuffled.

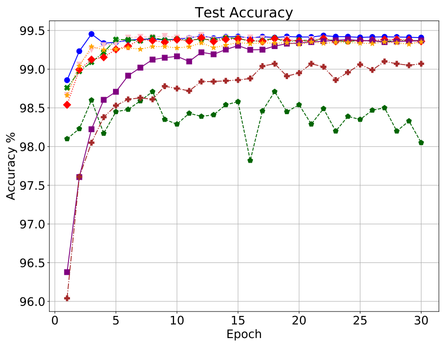
5.2 Accuracy Results

Evaluating the learning rate policies: Among the state-of-the art learning policies that we enhanced for CapsNets, *warm restarts* is the most promising with maximum accuracy improved of 0.074%. CapsNet with *warm restarts* reaches the maximum accuracy of the baseline (with fixed learning rate) in 6 epochs rather than 29, with a training time reduction of 62.07%.

Evaluating Adaptive batch size: Different combinations of batch sizes in *adaptive batch size* algorithm have been tested, since the smaller the batch size, the faster the initial convergence. However, large batch sizes lead to slightly higher accuracy after 30 epochs and, most importantly, to a reduced training time. In fact, a CapsNet training epoch with batch size 1 lasts for 7 minutes, while with batch size 128 it lasts for only 28 seconds. Batch size 16 is a good trade off between fast convergence and short training time (i.e., 49sec/epoch). The best results, applying *adaptive batch size*, are obtained using batch size 1 for the first three epochs and then increasing it to 16 for the remaining part of the training. With this parameter selection, there is a 0.044% accuracy gain with respect to baseline and the maximum accuracy of the baseline is reached in 5 epochs rather than in 29. However, the first three epochs take a longer time (88% longer time) because of the reduced batch



(a)



(b)

Figure 6: *Legend is in common for both figures.* **(a)** Comparison of different optimization types offered by our X-TrainCaps framework, on the basis of accuracy, training time and number of parameters. The training time is computed as the number of epochs to reach the maximum accuracy, multiplied by time (in seconds) per epoch. The words WAB and WS stand for *WarmAdaBatch* and *WeightSharing*, respectively, with *WeightSharing* including also the small-decoder optimization. **(b)** Accuracy evolution over the training epochs for different optimization solutions, showing a high accuracy improvement for many policies, with a high reduction in training time and high accuracy for the ones based on *warm restarts*.

size, so *adaptive batch size* alone is not convenient. However, the total training time is reduced by 30%, compared to the baseline.

Evaluating WarmAdaBatch: As for the batch, the first cycle of learning rate lasts 3 epochs and the second one 27 epochs. Variation of batch size and learning rate cycles are synchronized. This solution allows to have a 0.088% gain in accuracy with respect to CapsNet baseline implementation, and the baseline maximum accuracy is reached by CapsNet with *WarmAdaBatch* in 3 epochs against 29. After the first three epochs batch size changes and learning rate is restarted, so there is a drop of accuracy, that however re-converges in a few steps to the highest stable value obtained.

Evaluating Weight Sharing: applying *weight sharing* to the DigitCaps layer, we are able to achieve a 15% reduction in the number of total parameters, decreasing from 8.2 millions to 6.7 millions. However, this reductions leads also to a decrease in the maximum accuracy, dropping by 0.26%.

5.3 Comparison of different optimization types

We compare the different types of optimizations in terms of accuracy, and also based on the training time to reach the maximum accuracy and the number of parameters. As we can see in Figure 6a, we compare different optimization methods in a 3-dimensional space. This representation provides Pareto-optimal solutions, depending on the optimization goals. We also compare, in Figure 6b, the accuracy and the learning rate evolution in different epochs, for *AdaBatch*, *WarmRestarts* and *WarmAdaBatch*. Among the space of the solutions, we discuss in more details two Pareto-optimal ones, *WarmAdaBatch* and the combination of *WarmAdaBatch* and *weight sharing*, which we call WAB+WS.

WarmAdaBatch: This solution provides the optimal point in terms of accuracy and training time, because it achieves the highest accuracy (99.454%) in the shortest time (2400 seconds). Varying the batch size boosts the accuracy improvements in the first epoch and the restart policy contributes to speed-up the training.

WAB+WS: The standalone *weight sharing* reduces the number of parameters by 15%. By a combination of it with *WarmAdaBatch*, the accuracy loss is compensated (99.38% versus 99.366% of the baseline), while training time is shorter than the baseline (7200 seconds versus 8700 seconds) but longer than simple *WarmAdaBatch*. Our framework chooses this solution if also the number of parameter reduction is selected among the optimization goals.

6 Conclusion

In this paper, we proposed X-TrainCaps, a novel framework to automatically optimize the training of a given CapsNet for accuracy, training time or number of parameters, based on the requirements needed. We enhanced the different learning policies, for the first time, for accelerating the training of CapsNets. Afterwards, we discussed how an integrated training framework can be developed to find Pareto-optimal solutions, including different new optimizations for fast training like *WarmAdaBatch*, complexity reduction for the CapsNet decoder, and *weight sharing* for CapsNets. These solutions not only provide significant reduction in the training time while preserving or improving the accuracy, but also enable a new mechanism to provide trade-off between training time, network complexity, and achieved accuracy. This enables new design points under different user-provided constraints.

References

[1] Reza Babanezhad, Mohamed Osama Ahmed, Alim Virani, Mark W. Schmidt, Jakub Konecný, and Scott Sallinen. "Stop Wasting My Gradients: Practical SVRG". In: *CoRR* abs/1511.01942 (2015) (cit. on p. 3).

[2] Hadi Daneshmand, Aurélien Lucchi, and Thomas Hofmann. "Starting Small - Learning with Adaptive Sample Sizes". In: *CoRR* abs/1603.02839 (2016) (cit. on p. 3).

[3] Soham De, Abhay Kumar Yadav, David W. Jacobs, and Tom Goldstein. "Big Batch SGD: Automated Inference using Adaptive Batch Sizes". In: *CoRR* abs/1610.05792 (2016) (cit. on pp. 1, 3).

[4] Aditya Devarakonda, Maxim Naumov, and Michael Garland. "AdaBatch: Adaptive Batch Sizes for Training Deep Neural Networks". In: *CoRR* abs/1712.02029 (2017) (cit. on pp. 1, 3).

[5] Priya Goyal, Piotr Dollár, Ross B. Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour". In: *CoRR* abs/1706.02677 (2017) (cit. on pp. 1, 3).

[6] G. E. Hinton, A. Krizhevsky, and S. D. Wang. "Transforming Auto-encoders". In: *ICANN* (2011) (cit. on p. 2).

[7] G. E. Hinton, S. Sabour, and N. Frosst. "Matrix capsules with EM routing". In: *ICLR* (2018) (cit. on p. 2).

[8] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. *THE MNIST DATABASE of handwritten digits*. 1998. URL: `http://yann.lecun.com/exdb/mnist/` (cit. on pp. 2, 7).

[9] Ilya Loshchilov and Frank Hutter. "SGDR: Stochastic Gradient Descent with Restarts". In: *CoRR* abs/1608.03983 (2016) (cit. on pp. 1, 3).

[10] Adam Paszke et al. "Automatic differentiation in PyTorch". In: *NIPS-W*. 2017 (cit. on p. 7).

[11] S. Sabour, N. Frosst, and G. E. Hinton. "Dynamic Routing Between Capsules". In: *NIPS* (2017) (cit. on pp. 1, 2, 7).

[12] Leslie N. Smith. "A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay". In: *CoRR* abs/1803.09820 (2018) (cit. on pp. 1, 3).

[13] Leslie N. Smith. "No More Pesky Learning Rate Guessing Games". In: *CoRR* abs/1506.01186 (2015) (cit. on p. 3).