electronics

MDPI

Article

FasterAI: A Lightweight Library for Neural Networks Compression

Nathan Hubens [1,2,*], Matei Mancas [1], Bernard Gosselin [1], Marius Preda [2] and Titus Zaharia [2]

[1] ISIA Lab, University of Mons (UMONS), 31, Bd. Dolez, 7000 Mons, Belgium
[2] Institut Polytechnique de Paris, Télécom SudParis, Advanced Research and TEchniques for Multidimensional Imaging Systems Department, 9 rue Charles Fourier, 91000 Évry, France
* Correspondence: nathan.hubens@umons.ac.be

Abstract: FasterAI is a PyTorch-based library, aiming to facilitate the use of deep neural network compression techniques, such as sparsification, pruning, knowledge distillation, or regularization. The library is built with the purpose of enabling quick implementation and experimentation. More particularly, compression techniques are leveraging callback systems of libraries, such as fastai and Pytorch Lightning to propose a user-friendly and high-level API. The main asset of FasterAI is its lightweight, yet powerful, simplicity of use. Indeed, because it has been developed in a very granular way, users can create thousands of unique experiments by using different combinations of parameters, with only a single line of additional code. This allows FasterAI to be suited for practical usage, as it contains the most common compression techniques available out-of-the-box, but also for research, as implementing a new compression technique usually boils down to writing a single line of code. In this paper, we propose an in-depth presentation of the different compression techniques available in FasterAI. As a proof of concept and to better grasp how the library is used, we present results achieved by applying each technique on a ResNet-18 architecture, trained on CALTECH-101.

Keywords: sparse neural networks; pruning; knowledge distillation; PyTorch library

Citation: Hubens, N.; Mancas, M.; Gosselin, B.; Preda, M.; Zaharia, T. FasterAI: A Lightweight Library for Neural Networks Compression. *Electronics* **2022**, *11*, 3789. https://doi.org/10.3390/electronics11223789

Academic Editors: Ping-Feng Pai and Domenico Mazzeo

Received: 30 September 2022
Accepted: 14 November 2022
Published: 18 November 2022

1. Introduction

Over the last few years, deep neural networks have witnessed an important increase in their amount of parameters and computation. As a result, the memory footprint and inference time have severely hindered the deployment of such methods, especially in resource-constrained environments, such as embedded systems or mobile devices. For that reason, neural network compression has been the subject of extensive research recently. Although many theoretical research studies have been conducted on compression, the field still lacks convenient tools for (1) practical applications but also (2) research. Additionally, because of this lack of tools, there is no standard way of implementing new compression techniques, making the comparison with previous techniques more difficult [1]. To solve this issue, we propose FasterAI [2], an open-source library, released under an Apache-2.0 license and available at https://nathanhubens.github.io/fasterai (accessed on 20 September 2022). It also includes extensive documentation and several tutorials to help users become acquainted with the library.

1.1. Related Work

The research field of neural network compression has recently been extremely active, leading to lots of published ideas [3–5], but also to the release of their corresponding implementations [6,7]. However, the available implementations may operate on different deep learning libraries and be designed for different application cases, thus requiring extensive adaptation in order to compare them. As a result, the field of compression can appear overwhelming for researchers that are willing to implement new techniques and

compare them with current methods, but also to newcomers that desire to compress their neural network for a concrete application.

Several pieces of work have proposed solutions to that problem by creating libraries allowing to seamlessly implement compression techniques, such as PyTorch Pruning [8] and Sparse ML [9]. However, those are mainly concerned with sparsification, neglecting other compression techniques, such as knowledge distillation and regularization. Another library, Nervana Distiller [10], provides a more thorough compression toolset, but is intended primarily for research usage. Additionally, most of those libraries require to implement new compression techniques in a self-contained way, limiting the opportunities for extensive experiments. In FasterAI, we aim at reducing the need of custom implementation to its bare minimum. Indeed, implementing a new method in FasterAI usually boils down to writing a single line of code. Moreover, to the best of our knowledge, FasterAI is the first compression library available for both fastai [11] and PyTorch Lightning [12].

1.2. Overview

The objective of FasterAI is twofold: (1) allow users not familiar with the domain to apply compression techniques; and (2) allow researchers to easily implement new compression methods and perform various experiments. FasterAI is organized around four modules, each one providing distinct compression capabilities, and which might depend on several arguments, as represented in Figure 1.

Figure 1. Illustration of the design followed by FasterAI.

Sparsify. The first module is responsible for making sparse neural networks, either in a static way, when retraining cannot be considered, or in a dynamic way, using callback systems, thus occurring during the training of the neural network.

Distill. This module is in charge of knowledge distillation techniques, i.e., training with a teacher–student paradigm, where a large model guides a smaller one to reach better performance, thus compressing the knowledge of a large model into a smaller one.

Regularize. The regularize module handles group regularization methods, i.e., techniques adding a penalty term on the magnitude of the weights, acting as a feature selection method, where some weights will be pushed toward 0, leading to a learned sparse model.

Misc. The last module includes singular compression methods, such as batch normalization folding, removing batch normalization layers, which can be considered useless after the training phase. It also includes factorization methods for fully connected layers that replace large weight matrices with smaller ones, thus reducing the total amount of weight.

To summarize, with FasterAI, we provide:

- An extensive, documented and open-source PyTorch-based neural network compression library.
- A new granular design approach for compression techniques, allowing to seamlessly perform thousands of different compression methods, by simply choosing between available options.
- A framework suited for practical cases as well as for research, by providing common compression techniques available out-of-the-box and allowing the conception of new compression methods in a single line of code.

This paper is divided into four sections, each one describing a compression module of FasterAI. In particular, we want to highlight how convenient it is to perform different kinds of experiments, either using well-known techniques, or creating novel ones. Indeed,

by leveraging the callback system of recent deep learning libraries, such as fastai and Pytorch Lightning, FasterAI provides a user-friendly and high-level API, allowing to easily combine and customize compression techniques. Although FasterAI is suitable for many types of architecture, we illustrate its use for convolutional neural networks.

2. Sparsify

The core of FasterAI resides in its sparsify module, containing capabilities for creating sparse networks, i.e., networks in which a large number of weight values are zeroes. FasterAI possesses two main ways to create a sparse network: (1) the static way, by using the Sparsifier class, able to sparsify either a specified layer, or the whole model, (2) the dynamic way, by using the SparsifyCallback, that must be used in conjunction with training, and that removes weights while the network is learning. Examples of usage for both methods are expressed in Listing 1.

Listing 1. The two ways of sparsifying a model. The static is performed offline, disconnected from training, while the dynamic is performed during training.

```
# (1) Static
sp=Sparsifier(model, granularity, context, criteria)
sp.prune_model(sparsity)

# (2) Dynamic
sp_cb=SparsifyCallback(sparsity, granularity, context, criteria, schedule)
learner.fit(n_epochs, cbs=sp_cb)
```

While the static way is faster to apply, as it does not require any additional steps, the lack of retraining after the removal of some parameters deeply impairs the model performance. For that reason, the dynamic way is most of the time preferred when trying to achieve compression while keeping the performance as high as possible. Although the distinction is not always clear in the literature, we make the difference within FasterAI between the process of sparsification, i.e., making neural network weights sparse, and pruning, i.e., physically removing those sparse weights. Indeed, the SparsifyCallback does not allow to remove any network's weight but rather to create a binary mask, of the same structure as the weights, and applies it to either sparsify a weight (when the mask value is 0) or keep it unchanged (when the mask value is 1). Weights sparsified during such a process are still present in the computation graph but do not participate in the final decision anymore.

The whole power of the sparsification capabilities of FasterAI lies in its SparsifyCallback, designed around four independent building blocks: granularity, context, criteria, and schedule, which are sufficient to fully describe the most common sparsification techniques. Those building blocks correspond to the four main axes of research in the field, each providing an answer to the following questions:

- Granularity: how to sparsify?
- Context: where to sparsify?
- Criteria: what to sparsify?
- Schedule: when to sparsify?

The purpose is to decompose the sparsifying problem into four subproblems. By doing so, each argument can be modified independently from the others, which allows to (1) create a vast number of opportunities and combinations for experiments and, (2) provide a unique and versatile callback, reducing the problem of implementing a novel sparsification technique to the modification of a single argument.

2.1. Granularity: How to Sparsify?

In FasterAI, the granularity designates the structure of the blocks of weights that are removed during the sparsification process. FasterAI handles most common sparsifying

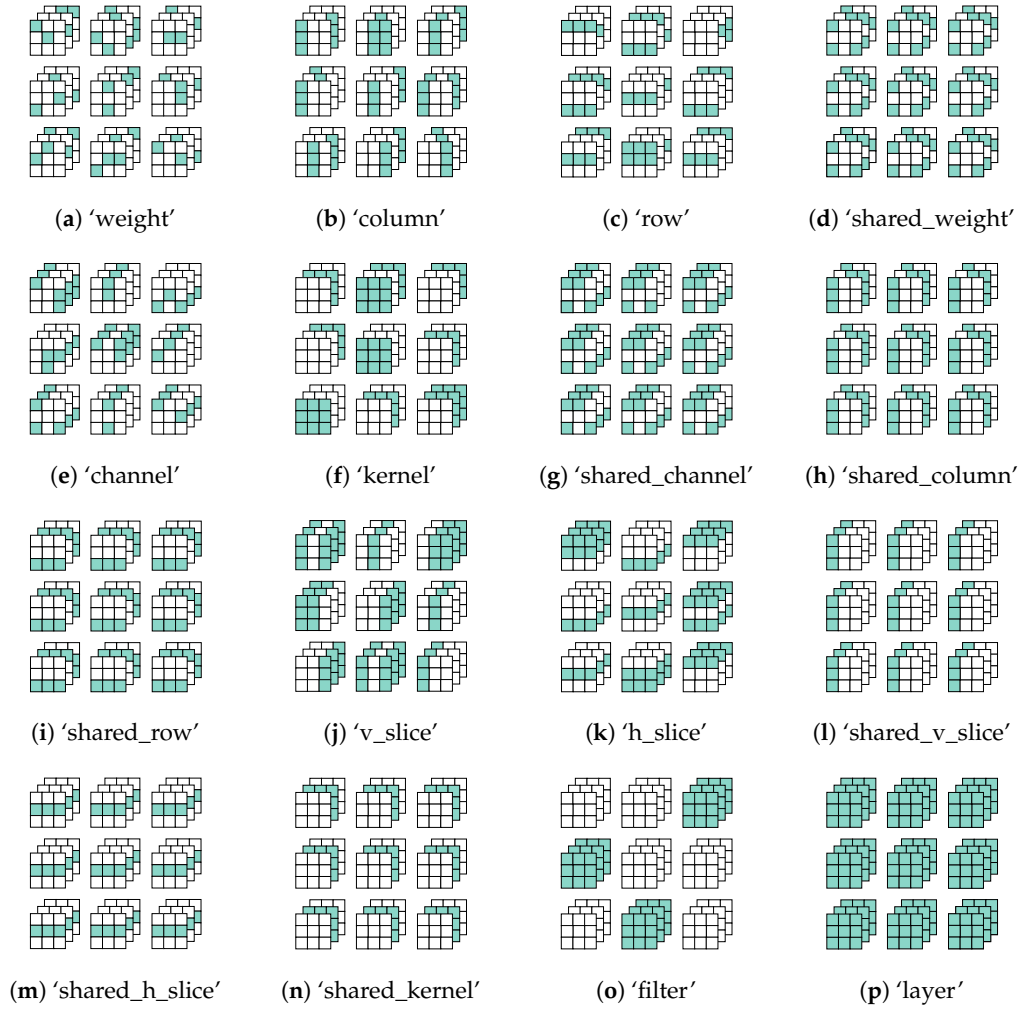
granularities, e.g., weight, kernel and filter, but also allows the use of more seldom ones, e.g., horizontal slices and shared kernels. In the literature, the terms unstructured and structured sparsity are often used to designate when sparsity is applied on weights (unstructured) or larger blocks (structured). In FasterAI, we adopt a more nuanced approach by defining as many granularities as there are slicing combinations of the weight tensor. In the case of 2D convolutions, 16 granularities are thus available by default. By following PyTorch conventions [8], the weights of a 2D convolutional layer are given by a 4D tensor of dimension [I, O, Kx, Ky], with I, O being, respectively, the input and output dimensions, and Kx, Ky, the dimensions of the convolutional kernel. The granularities available by default are defined by Listing 2.

Listing 2. Different available granularities for 4D weight tensor of dimension [I, O, Kx, Ky].

```
Weight                  (0D) = Weights[i, o, kx, ky]
Column                  (1D) = Weights[i, o, kx, : ]
Row                     (1D) = Weights[i, o, : , ky]
Shared-Weight           (1D) = Weights[i, :, kx, ky]
Channel                 (1D) = Weights[:, o, kx, ky]
Kernel                  (2D) = Weights[i, o, : , : ]
Shared-Channel          (2D) = Weights[:, :, kx, ky]
Shared-Column           (2D) = Weights[i, :, kx, : ]
Shared-Row              (2D) = Weights[i, :, : , ky]
Vertical-Slice          (2D) = Weights[:, o , : ,ky]
Horizontal-Slice        (2D) = Weights[:, o , :, ky]
Shared-Vertical-Slice   (3D) = Weights[:, :, kx, : ]
Shared-Horizontal-Slice (3D) = Weights[:, :, : , ky]
Shared-Kernel           (3D) = Weights[i, :, : , : ]
Filter                  (3D) = Weights[:, o, : , : ]
Layer                   (4D) = Weights[:, :, : , : ]
```

These granularities are represented in Figure 2, sorted by "how structured" the granularity is. On top of the presented granularities, suited for ConvNets, FasterAI also proposes granularities for fully connected Layers, as well as for self-attention layers, required in the transformers' architectures. FasterAI allows a wide variety of granularities, along which the network's parameters will be sparsified. Among less common granularities, we introduce the concept of "shared granularity", indicating granularity structures that are shared between all filters. For example, shared_weight in Figure 2d defines a granularity, where weights are selected individually in each filter, but the same selection pattern is applied to each filter in the layer.

As a proof of concept, we conduct an experiment to highlight the impact of pruning granularity on the performance of a neural network. We choose the ResNet-18 architecture [13], as it is a model commonly used for pruning benchmarking, and apply it to the CALTECH-101 dataset [14], various in images and classes, that is split using a 80:20 split between training and validation sets. The model is trained for 30 epochs, using a learning rate value of 1×10^{-3}, the Adam [15] optimizer, and a batch size of 64. We then compare the final validation accuracy, obtained after sparsifying with each of the available granularities. There are 4 sparsity levels that are studied: 30%, 50%, 70% and 90%. Two initialization methods are considered: either the model is trained from scratch, i.e., the weights are randomly initialized, or finetuned from a pretrained version. The context, criteria and schedule are respectively set to local, large_final and one_cycle. The results, as well as the baseline, i.e., the unpruned model, are presented in Table 1. For readability constraints, the name of the granularities in Table 1 are abbreviated, e.g., s-v-slice corresponds to the shared_vertical_slice granularity. The layer sparsity is voluntarily omitted, as it is not available in the local context.



(**a**) 'weight' (**b**) 'column' (**c**) 'row' (**d**) 'shared_weight'

(**e**) 'channel' (**f**) 'kernel' (**g**) 'shared_channel' (**h**) 'shared_column'

(**i**) 'shared_row' (**j**) 'v_slice' (**k**) 'h_slice' (**l**) 'shared_v_slice'

(**m**) 'shared_h_slice' (**n**) 'shared_kernel' (**o**) 'filter' (**p**) 'layer'

Figure 2. Common granularities, available in FasterAI. Weights are arranged in 9 filters of 3 channels and dimension 3×3, where colored weights are sparsified according to the chosen granularity.

Table 1. Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to local, large_final and one_cycle. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Scratch				**Finetune**			
	30%	**50%**	**70%**	**90%**	**30%**	**50%**	**70%**	**90%**
Baseline	80.61 ± 0.42				90.03 ± 0.54			
weight	80.40 ± 0.22	80.20 ± 0.71	79.74 ± 0.85	78.46 ± 0.40	91.39 ± 0.14	91.67 ± 0.56	91.43 ± 0.10	88.75 ± 0.90
column	81.04 ± 0.46	80.58 ± 0.15	80.34 ± 0.22	75.60 ± 0.50	91.85 ± 0.59	91.23 ± 0.34	90.43 ± 0.47	83.55 ± 1.53
row	81.13 ± 1.01	80.18 ± 0.56	79.80 ± 0.19	75.97 ± 0.65	91.56 ± 0.79	90.57 ± 0.54	90.88 ± 1.06	85.12 ± 0.97
s-weight	80.49 ± 0.69	80.14 ± 0.19	78.57 ± 1.03	66.23 ± 0.97	90.83 ± 1.20	90.06 ± 0.25	88.40 ± 0.67	46.97 ± 0.87
channel	81.29 ± 0.43	79.50 ± 0.62	79.81 ± 0.63	65.85 ± 0.32	91.16 ± 0.41	90.34 ± 1.03	86.60 ± 0.43	44.97 ± 0.62
kernel	80.27 ± 0.49	79.50 ± 0.51	79.52 ± 1.52	67.34 ± 1.57	91.79 ± 0.43	91.25 ± 0.43	89.88 ± 0.73	77.64 ± 0.80
s-channel	79.54 ± 0.87	80.85 ± 0.98	78.21 ± 0.56	38.60 ± 3.12	90.75 ± 0.55	90.55 ± 0.52	86.20 ± 1.34	23.03 ± 0.92
s-column	79.69 ± 0.64	79.87 ± 0.23	80.23 ± 0.40	59.90 ± 1.71	90.70 ± 0.69	89.93 ± 0.31	86.00 ± 0.32	41.28 ± 3.53
s-row	80.51 ± 0.34	79.78 ± 0.29	77.92 ± 0.56	57.99 ± 0.89	90.30 ± 0.51	89.88 ± 0.53	85.38 ± 0.57	40.43 ± 0.92
v slice	80.01 ± 0.85	80.43 ± 0.91	78.28 ± 0.35	48.09 ± 1.55	90.59 ± 0.45	89.46 ± 0.71	82.90 ± 0.30	29.81 ± 1.78
h slice	80.22 ± 0.79	79.19 ± 0.82	77.13 ± 1.60	43.95 ± 1.02	91.01 ± 0.87	88.75 ± 0.36	82.48 ± 0.96	29.60 ± 2.35
s-v-slice	79.91 ± 0.84	80.40 ± 0.50	76.97 ± 0.47	74.98 ± 0.92	89.28 ± 0.35	89.17 ± 0.62	75.47 ± 0.75	73.49 ± 1.09
s-h-slice	79.81 ± 0.76	80.07 ± 0.64	76.22 ± 0.11	74.58 ± 0.35	89.77 ± 0.22	88.60 ± 0.58	73.41 ± 1.21	69.15 ± 0.77
s-kernel	79.96 ± 0.11	81.00 ± 0.21	75.84 ± 1.60	48.56 ± 0.94	89.77 ± 0.43	89.30 ± 0.48	81.25 ± 0.27	31.86 ± 1.23
filter	80.76 ± 1.65	78.41 ± 1.01	72.92 ± 2.40	37.47 ± 1.90	89.86 ± 0.85	87.78 ± 0.18	77.52 ± 1.84	29.05 ± 0.09

From those results, it can be observed that the general trend is that the more structured granularity is, i.e., the larger the size of structures removed, the larger the drop in final

performance. This can be explained by the fact that more structured granularities are less precise in their selection of weights, potentially removing weights that might be important for the network. It is, however, important to mention that, although less performant, more structured granularities allow for an easier speed-up in practice, as they require less overhead to store sparse weight indices [4]. Additionally, it can be observed that smaller granularities and low sparsity levels can lead to better performance than the baseline, illustrating the regularization capabilities of sparsification, thus helping to reduce overfitting and increase generalization.

2.2. Context: Where to Sparsify?

In FasterAI, the context refers to the locality of the selection of the weights. In the literature, the two most common options are: (1) local pruning, i.e., the selection of the weights is performed in each layer separately, producing equally sparse layers in the network, and (2) global pruning, i.e., the selection of the weights is performed by comparing those of the whole network, producing a network with different sparsity levels for each layer. Both techniques are expressed in a simplified way in Listing 3.

Listing 3. Simplified representation of local sparsification, comparing weights in each layer independently and global sparsification, comparing weights from all the layers.

```
# (1) Local
for layer in layers:
mask = compute_mask(layer.weight, sparsity)
pruned_model = prune_layer(layer, mask)

# (2) Global
global_weights = concat[(layer.weight) for layer in layers]
mask = compute_mask(global_weights, sparsity)
pruned_model = prune_model(mask)
```

FasterAI handles both methods by default, only by selecting the local or global method accordingly in the SparsifyCallback. Local and global sparsification have different implications on the final sparsity of the network, with local context leading to equally sparse layers in the network and global context leading to layers with differences in sparsities, which can pose issues for networks possessing bottlenecks, where it can be undesirable to remove too many parameters. In the case that the user wants to specify a particular sparsity level for certain layers, FasterAI accepts a list of sparsities that will be applied to corresponding layers.

We propose to compare the impact of each context on the performance of a neural network. For this purpose, we use the same architecture, datasets and training parameters as the experiment conducted in Subsection 2.1 but using a global context instead. The results are provided in Table 2. For readability constraints, the same abbreviations are applied to the name of granularities in Table 2.

As can be observed in Table 2, the general trend seems to be that more coarse granularities perform worse than more precise ones. Additionally, the drop in performance for high sparsities is larger when the network has been fine-tuned than when trained from scratch. By comparing Tables 1 and 2, it can be observed that global sparsifying seems to achieve better results in the scratch training regime, while providing similar results when fine-tuning.

Table 2. Results of sparsifying ResNet-18 for all available granularities. Context, criteria and schedule are respectively set to global, large_final and one_cycle. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Scratch				Finetuned			
	30%	**50%**	**70%**	**90%**	**30%**	**50%**	**70%**	**90%**
weight	80.58 ± 1.21	80.42 ± 0.73	80.31 ± 1.01	79.63 ± 0.59	91.12 ± 0.59	91.17 ± 0.19	91.79 ± 0.35	89.90 ± 0.14
column	80.51 ± 0.58	80.78 ± 0.48	80.03 ± 1.08	78.87 ± 0.85	91.54 ± 0.50	90.90 ± 0.46	91.43 ± 0.61	87.48 ± 0.44
row	81.13 ± 1.30	80.65 ± 0.71	79.50 ± 1.52	78.57 ± 0.45	91.19 ± 0.88	91.72 ± 0.26	91.32 ± 0.90	87.78 ± 0.48
s-weight	79.96 ± 0.57	80.45 ± 0.30	76.84 ± 0.72	58.92 ± 3.78	90.99 ± 0.23	90.96 ± 0.68	84.65 ± 1.28	35.83 ± 1.12
channel	80.78 ± 0.52	80.53 ± 0.12	77.79 ± 1.31	63.91 ± 0.74	91.27 ± 0.22	90.55 ± 0.38	80.91 ± 0.70	18.29 ± 1.88
kernel	81.89 ± 0.39	80.49 ± 0.29	79.74 ± 0.41	72.08 ± 1.91	91.08 ± 0.37	91.05 ± 0.52	90.43 ± 0.36	85.25 ± 0.66
s-channel	79.01 ± 1.33	78.70 ± 0.72	73.54 ± 0.92	72.87 ± 1.10	89.00 ± 0.35	85.65 ± 1.72	74.03 ± 1.52	61.82 ± 1.69
s-column	80.87 ± 0.93	79.74 ± 0.39	75.78 ± 0.54	68.85 ± 3.59	90.28 ± 0.22	89.26 ± 0.51	72.14 ± 0.52	18.47 ± 5.31
s-row	80.36 ± 0.87	80.80 ± 0.67	74.58 ± 1.68	68.31 ± 1.23	91.21 ± 0.45	89.57 ± 0.11	65.55 ± 4.13	23.01 ± 6.27
v slice	80.34 ± 0.99	78.72 ± 0.54	75.09 ± 0.39	45.40 ± 4.10	90.81 ± 0.62	87.98 ± 0.64	76.68 ± 0.66	25.36 ± 1.99
h slice	80.36 ± 0.67	79.83 ± 0.34	75.47 ± 0.22	47.52 ± 1.77	90.85 ± 0.41	87.96 ± 0.34	78.46 ± 0.62	27.66 ± 2.97
s-v-slice	79.54 ± 0.36	78.67 ± 0.27	78.14 ± 1.03	78.39 ± 0.90	90.04 ± 0.43	87.56 ± 0.86	85.45 ± 0.40	81.89 ± 0.85
s-h-slice	79.72 ± 0.23	79.19 ± 0.45	78.79 ± 0.49	78.36 ± 0.79	89.30 ± 0.42	86.49 ± 1.05	85.39 ± 0.43	83.39 ± 0.47
s-kernel	79.18 ± 0.56	76.79 ± 1.32	75.64 ± 1.47	75.05 ± 1.64	86.34 ± 0.47	84.76 ± 1.43	75.33 ± 6.07	25.60 ± 2.26
filter	79.36 ± 1.22	78.52 ± 0.65	72.83 ± 1.44	68.09 ± 4.39	90.39 ± 0.89	84.43 ± 0.11	62.64 ± 1.58	44.15 ± 1.64

2.3. Criteria: What to Sparsify?

The criteria are a fundamental component of any sparsifying technique, as they act as a proxy for weight importance. In practice, applying the desired criteria to each group of weights returns a score, according to which the selection of weights is based. Group of weights with the lowest score will be zeroed out first, while those having the largest will be retained. There exist many sparsifying criteria [16], with 14 currently available by default in FasterAI, and expressed in a simplified way, following PyTorch notation, in Listing 4. To that end, we define wi and wf, respectively, being the initial and final values of the weights, i.e., their values at the initialization and at the current step of training.

Listing 4. The list of criteria available in FasterAI and their PyTorch implementation.

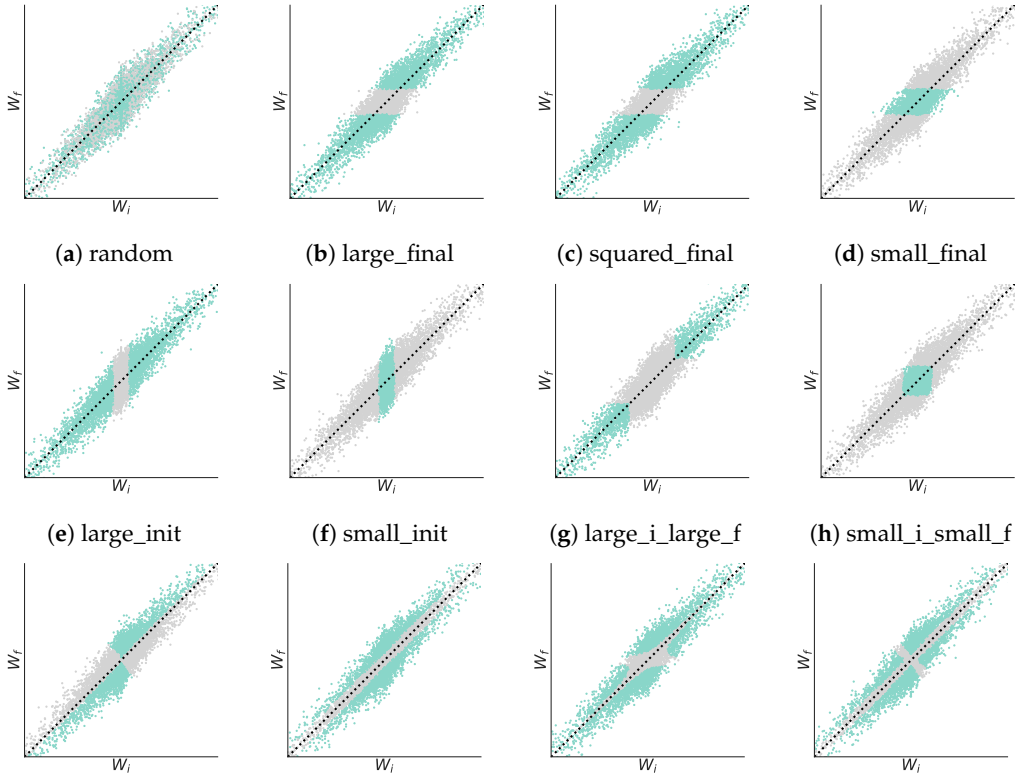
```
random                 = torch.randn_like(wf)
large_final            = torch.abs(wf)
squared_final          = torch.square(wf)
small_final            = torch.neg(torch.abs(wf))
large_init             = torch.abs(wi)
small_init             = torch.abs(torch.neg(wi))
large_init_large_final = torch.abs(torch.min(wf, wi))
small_init_small_final = torch.abs(torch.neg(torch.max(wf, wi)))
magnitude_increase     = torch.sub(torch.abs(wf),torch.abs(wi))
movement               = torch.abs(torch.sub(wf,wi))
```

Because of the way the criteria are implemented in FasterAI, it is very convenient to create custom criteria. Indeed, implementing new selection criteria boils down to writing a single function that will be applied to each weight before computing the sparsification mask to be applied. For example, we introduce a novel criterion named mov_large_final, which is similar to the movement one, but puts more emphasis on the final value of weights. Similarly, we introduce another criterion, named mov_mag, which considers weights whose absolute value has moved the most. Those criteria are expressed in Listing 5.

Listing 5. Custom criteria and their corresponding implementation in PyTorch.

```
mov_large_final = torch.abs(torch.mul(x, torch.sub(x,y))))
mov_mag         = torch.abs(torch.sub(torch.abs(x),torch.abs(y))))
```

The decision boundaries of available criteria are represented in Figure 3. In this figure, we represent the weight distribution at initialization, W_i, against their value at the current training step, W_f.



(**a**) random (**b**) large_final (**c**) squared_final (**d**) small_final

(**e**) large_init (**f**) small_init (**g**) large_i_large_f (**h**) small_i_small_f

(**i**) magnitude_increase (**j**) movement (**k**) mov_large_final (**l**) mov_mag

Figure 3. Common pruning selection criteria. A schematic weight distribution is represented, where colored weights are considered important by the criteria, while greyed out ones are removed.

In practice, at each sparsifying phase, the chosen criteria are applied to each weight, before aggregating them according to the desired granularity. The pruning mask is computed by retaining the weights having the largest score, according to the desired context and sparsity level. It is then applied to replace weights considered less important by the criteria by zeroes. Additionally, FasterAI keeps track of the values of the weights during training. This paves the way to creating criteria using first-order information, taking the training dynamics into account.

In Table 3, we report the comparison between all the available criteria. Experiments are conducted in the same conditions as for the previous experiments, and with the same architecture and dataset. The granularity is set to weight, the context to local, and the schedule to one_cycle. For readability constraints, the names of criteria in Table 3 are abbreviated, e.g., large i,f corresponds to the large_i_large_f criteria.

From those results, we can observe that the criteria has a minor effect on the performance at low sparsity level, e.g., 30%. This can be explained by the fact that the network, although having a part of parameters that are removed, still possesses enough capacity to compensate for the removed weights and achieve decent performance. When the sparsity level increases, however, criteria based on lower weight values, e.g., small f, small i, small i,f, seem to perform badly. This phenomenon happens because weights with low values do not participate much in the final results, and thus are not holding much discriminative information about the data.

Table 3. Results of sparsifying ResNet-18 for all available criteria. Granularity, context and schedule are respectively set to weight, local and one_cycle. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Scratch				Fine-Tune			
	30%	50%	70%	90%	30%	50%	70%	90%
random	80.53 ± 0.70	80.38± 0.23	80.01 ± 0.94	66.34 ± 0.92	90.24 ± 1.47	90.32 ± 0.55	86.38 ± 0.80	30.23 ± 4.25
large f	80.03 ± 1.07	80.93± 0.97	80.82 ± 0.48	77.99 ± 0.72	91.41 ± 0.43	91.17 ± 0.32	91.41 ± 1.08	88.09 ± 0.45
small f	79.05 ± 0.66	75.42± 0.44	48.49 ± 1.42	19.38 ± 3.42	84.37 ± 0.61	74.20 ± 0.50	18.76 ± 0.27	0.91 ± 0.52
sq f	80.76 ± 0.71	79.34± 1.10	79.58 ± 0.58	78.61 ± 0.23	91.39 ± 0.35	90.99 ± 0.30	90.97 ± 0.12	88.97 ± 0.92
large i	80.42 ± 0.85	79.67± 0.44	79.61 ± 0.73	75.62 ± 0.25	91.56 ± 0.52	92.58 ± 0.49	92.34 ± 1.13	90.30 ± 0.49
small i	80.67 ± 0.90	80.78± 1.19	79.60 ± 0.74	73.19 ± 0.50	86.52 ± 0.09	85.61 ± 1.03	83.42 ± 1.20	0.88 ± 0.31
large i,f	80.03 ± 0.43	80.07± 0.54	79.18 ± 0.57	72.37 ± 0.78	92.21 ± 0.43	92.43 ± 0.89	91.74 ± 0.12	88.00 ± 0.67
small i,f	80.25 ± 0.97	76.39± 0.83	65.50 ± 0.76	23.27 ± 1.66	85.23 ± 0.83	76.75 ± 1.07	31.27 ± 1.31	3.48 ± 3.88
mag inc	80.40 ± 0.44	80.27± 0.34	79.72 ± 0.16	79.19 ± 1.99	90.83 ± 0.92	90.06 ± 0.52	88.88 ± 0.38	83.13 ± 0.61
mov	80.45 ± 0.63	79.98± 0.43	80.20 ± 0.39	78.26 ± 0.37	89.73 ± 0.19	89.99 ± 0.43	88.58 ± 0.63	80.56 ± 1.23
mov_large_f	80.16 ± 1.16	82.24± 0.67	81.47 ± 0.38	79.12 ± 0.14	90.90 ± 0.14	90.96 ± 0.99	89.93 ± 0.22	86.18 ± 0.48
movmag	80.05 ± 0.37	80.99 ± 0.97	80.32 ± 0.61	79.69 ± 0.57	90.12 ± 1.52	90.55 ± 0.83	90.06 ± 0.34	88.22 ± 0.25

2.4. Schedule: When to Sparsify?

The last argument required in the SparsifyCallback is the sparsification schedule. It defines when the sparsification process will occur during the training phase. Traditionally, the most common schedules are the one-shot, which performs the sparsification in a single step, and iterative, which performs it in several steps. Those methods usually required a fine-tuning phase after each sparsification stage to help the network to recover from the lost performance. In FasterAI, all schedules are implemented within a single class, the only differentiation being defined according to three parameters:

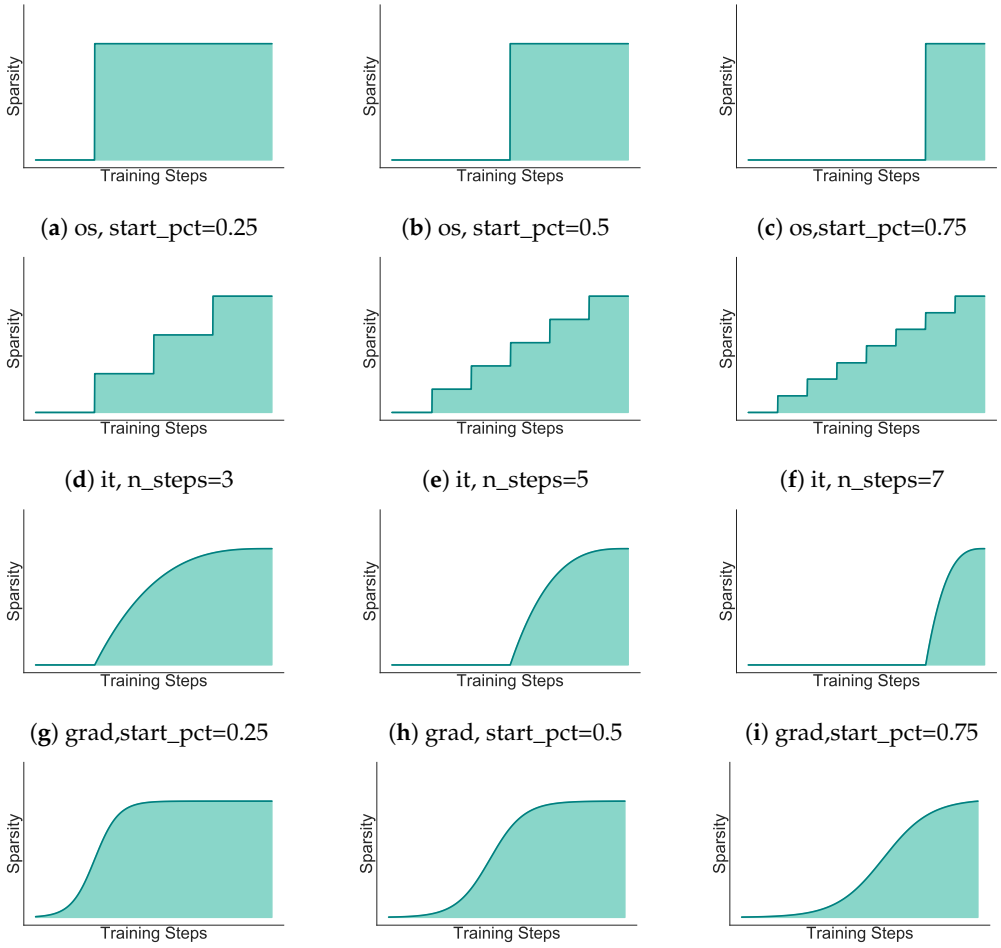
- start_pct (default to 0): the percentage of training at which the sparsification process starts, i.e., for how long the model will be pretrained.
- end_pct: the percentage of training at which the sparsification process stops, i.e., for how long the model will be fine-tuned after being sparsified.
- schedule_function: the function describing the evolution of the sparsity during the training. There are four currently available by default: one_shot, iterative, gradual [17], and one_cycle [18]. Those schedule functions are expressed in Listing 6.

Listing 6. Schedules available by default in FasterAI and their corresponding implementation.

```
def one_shot(sparsity, t_step): return~sparsity

def iterative(sparsity, t_step, n_steps=5):
return  (sparsity/n_steps)*(np.ceil((t_step)*n_steps))

def gradual(sparsity, t_step): return sparsity * (1 - t_step)**3

def one_cycle(sparsity, t_step, α=14, β=6):
return (1+np.exp(-α+β)) /(1 +(np.exp(-α*t_step+β)))*sparsity
```

By shifting the complexity of the pruning schedule to the schedule_function, we ensure that all schedules can be defined in FasterAI. By doing so, we remove the need for complex training loops, as all schedules are applied in a single main training phase.

In Figure 4, we represent variations of the four available sparsifying schedules, where adjustments are made to customize the schedule behavior. As can be observed, the start_epoch and end_epoch can further help the user to alter the pruning schedule as desired. For example, in Figure 4b, the one-shot pruning schedule could also be used with a value of start_pct=0, becoming what is more well-known as pruning at initialization [19], achieving the target amount of sparsity right from the start of training. For readability constraints, we abbreviate the names of our schedules, e.g., one_shot becomes os.



(**a**) os, start_pct=0.25 (**b**) os, start_pct=0.5 (**c**) os,start_pct=0.75

(**d**) it, n_steps=3 (**e**) it, n_steps=5 (**f**) it, n_steps=7

(**g**) grad,start_pct=0.25 (**h**) grad, start_pct=0.5 (**i**) grad,start_pct=0.75

(**j**) ocp, α=20, β=4 (**k**) ocp, α=15, β=5 (**l**) ocp, α=10, β=6

Figure 4. Evolution of sparsity along training for the available pruning schedules. While the sched_func parameters defines the general evolution, the schedule can further be customized by modifying the start_pct and end_pct values.

We report in Table 4 the results of applying the schedules represented in Figure 4. Experiments were conducted in the same training conditions as previous ones. The granularity was set to weight, context to local and criteria to large_final. For readability constraints, in Table 4, the name of the schedule directly refers to the subfigure index in Figure 4.

Table 4. Results of sparsifying ResNet-18 for all available criteria. Granularity, context and criteria are respectively set to weight, local and large_final. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Scratch				**Finetune**			
	30%	**50%**	**70%**	**90%**	**30%**	**50%**	**70%**	**90%**
(a)	79.60 ± 1.02	79.69 ± 0.69	79.29 ± 0.38	77.33 ± 0.25	91.47 ± 0.54	91.48± 0.27	91.63 ± 0.13	91.08 ± 0.34
(b)	80.32 ± 0.95	81.02 ± 0.39	80.31 ± 0.31	78.59 ± 0.94	91.99 ± 0.56	91.28± 0.18	91.65 ± 0.36	90.55 ± 0.73
(c)	80.58 ± 0.73	80.12 ± 0.34	80.49 ± 0.61	79.45 ± 1.00	91.90 ± 0.74	90.94± 0.48	91.34 ± 0.57	89.28 ± 0.34
(d)	80.73 ± 0.14	80.31 ± 0.71	80.31 ± 0.43	77.94 ± 1.05	91.59 ± 0.94	91.28± 0.25	91.32 ± 0.20	88.42 ± 0.18
(e)	80.65 ± 1.12	80.49 ± 1.07	81.33 ± 0.80	72.39 ± 1.34	90.86 ± 0.18	90.65± 0.50	90.34 ± 0.63	85.47 ± 0.67
(f)	81.05 ± 0.67	81.05 ± 0.52	79.80 ± 0.72	69.91 ± 3.01	90.90 ± 0.34	90.63± 0.34	91.08 ± 0.31	80.94 ± 0.57
(g)	80.78 ± 0.55	80.01 ± 0.41	80.65 ± 0.69	79.54 ± 0.83	90.96 ± 0.43	90.43± 0.65	91.52 ± 0.57	90.94 ± 0.49
(h)	80.31 ± 0.86	81.24 ± 0.58	80.47 ± 0.54	79.96 ± 0.42	91.34 ± 0.28	91.05± 0.97	90.96 ± 0.41	89.72 ± 0.47
(i)	81.25 ± 1.18	81.35 ± 0.88	80.65 ± 1.21	78.26 ± 0.94	90.86 ± 0.69	91.36± 0.43	91.25 ± 0.66	87.24 ± 0.65
(j)	80.56 ± 0.76	80.74 ± 0.26	81.27 ± 0.69	79.60 ± 0.12	91.45 ± 0.53	91.89± 0.05	92.01 ± 0.65	91.87 ± 0.22
(k)	80.74 ± 0.27	80.11 ± 0.86	79.08 ± 0.35	78.83 ± 0.38	91.58 ± 0.50	91.94± 0.56	92.03 ± 0.85	90.61 ± 0.45
(l)	80.32 ± 0.36	81.25 ± 0.33	80.80 ± 0.43	79.46 ± 0.62	91.58 ± 0.23	90.83± 0.19	91.87 ± 0.51	88.13 ± 0.20

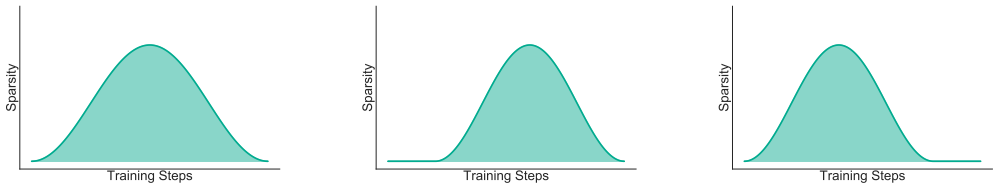
As can be observed, schedules implying a weight removal later in training seem to produce suboptimal results, especially in the fine-tuning regime. Indeed, removing parameters close to the end of training does not let enough time for the network to adjust its remaining weights to accommodate its weight loss. Additionally, schedules producing a gradual increase in sparsity, such as the gradual and one-cycle, seem to provide better and more stable results.

By modifying the three schedule parameters, users can also create their own pruning schedule or easily implement other existing ones, such as the dense–sparse–dense (DSD) schedule [20] for example, which increases the sparsity for the first half of training, then gradually decay it until the network is 0% sparse again. The corresponding schedule_function would be defined as in Listing 7.

Listing 7. Implementation of the dense–sparse–dense technique in FasterAI.

```
def dsd(sparsity, t_step):
if t_step<0.5: return (1 + math.cos(math.pi*(1-t_step*2))) * sparsity / 2
else: return (1 - math.cos(math.pi*(1-t_step*2))) * sparsity / 2
```

By then modifying the values of start_pct and end_pct in the SparifyCallback, we can further customize our pruning schedule, as displayed in Figure 5. Such a schedule_function also shows that it is possible not only to use a schedule to perform sparsification, but also weight growing, i.e., start from a sparse network, and gradually allow zeroed-out weights to be retrained, creating new connections in the network.



(**a**) dsd, start_pct=0, end_pct=1 (**b**) dsd, start_pct>0, end_pct=1 (**c**) dsd, start_pct=0,end_pct<1

Figure 5. Variation of the dsd schedule. The use of start_epoch and end_epoch help to further customize a given pruning schedule.

2.5. Lottery Ticket Hypothesis

Recent studies have demonstrated that an optimal sparse network could be discovered right from the initialization of a neural network, i.e., without any training being required [21,22]. This particularity is named the lottery ticket hypothesis (LTH) and was empirically demonstrated for simple datasets and architectures [21]. The optimal subnetwork is thus said to have "won" at the initialization lottery and is consequently named the "winning ticket". To generalize the concept to more complex cases, authors had to weaken the hypothesis, not extracting the optimal network from initialization anymore, but after a few iterations of training. This generalized method is called Lottery Ticket Hypothesis with Rewinding (LTHR) and the found subnetworks named "matching tickets" [23]. To discover such subnetworks, authors proposed to go through a five-step experiment, represented in Figure 6 and detailed below:

1. Train a freshly initialized network (W_0) for t iterations and save its set of weights (W_t).
2. Continue the training until completion (W_T).
3. Apply a pruning mask according to the desired sparsity level, granularity, context, criteria ($m \odot W_T$).
4. Reset the weights to their saved values, still applying the pruning mask ($m \odot W_t$).
5. Continue training and repeat the previous steps, each time updating the mask until the desired sparsity is achieved.

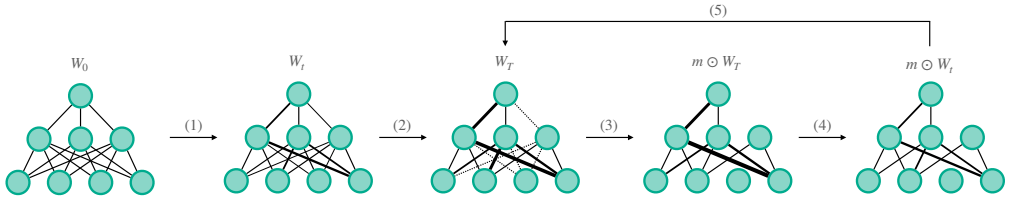


Figure 6. Illustration of the lottery ticket experiment, consisting of 5 steps, and providing the so-called winning (or matching) ticket.

In the case of the original LTH experiment being applied, the rewinding iteration t at which the set of weights is saved is equal to 0. For those experiments, authors used to sparsify their network according to the weights, globally, using the l_1-norm criteria, and following an iterative schedule. FasterAI handles such LTH experiments by default but allows to expand them to any granularities, contexts, criteria and schedules, opening the way to many novel experiments about finding winning tickets. To accomplish such a procedure in FasterAI, some additional arguments can be provided to the SparsifyCallback:

- lth: whether weights are reinitialized to their saved value after each pruning round.
- rewind_epoch (default to 0): the epoch of training where weights values are saved for further reinitialization.
- reset_end: whether to reset the weights to their saved values after training.

The classic Lottery Ticket Experiments [21,23] can be performed with Listing 8.

Listing 8. Changes to SparsifyCallback to perform lottery tickets experiments.

```
# Classic LTH
SparsifyCallback(sp,'weights','global', large_final, iterative, lth=True)

# LTH with Rewinding
SparsifyCallback(sp, 'weights', 'global', large_final, iterative, lth=True,\
rewind_epoch=1)
```

In Table 5, we report the results obtained when performing the classic LTH and LTHR techniques using the same architecture and datasets as the previous experiments. Each pruning round is performed for 30 epochs and the rewind_epoch is set to 1 for LTHR. We can observe that, in our case, both techniques provide similar results. Additionally, results show that it is possible to find high-performing pruned networks, even for high sparsity levels.

Table 5. Results of performing LTH and LTHR experiments on ResNet-18 trained on CALTECH-101. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

LTH				LTH with Rewinding			
30%	**50%**	**70%**	**90%**	**30%**	**50%**	**70%**	**90%**
80.05 ± 0.72	81.86 ± 0.40	84.59± 0.09	84.79± 0.37	80.65 ± 1.02	82.97± 1.15	83.92± 0.49	84.77± 0.46

2.6. Prune

As described previously, sparsification is usually introduced by applying a binary mask, multiplying the value to keep by 1, and those to remove by 0. This leads to a sparse network, difficult to accelerate in practice. However, some particular granularities allow the sparse weights to be physically removed from the network, effectively taking advantage of the compression to witness speed-up without any dedicated resource. Two granularities allow to perform such a feature: (1) filter and (2) shared-kernel.

Once a filter is completely zeroed out, it can be removed from the network, leading to a dense but smaller architecture. There is one subtlety, however, as removing the zeroed

filter is not enough for the architecture to be operational. When removing a filter, it changes the output shape of the concerned layer, as there is one less feature map. This means that the following convolutional layer now receives an input with fewer channels and thus, in all of its filters, the kernel corresponding to the removed feature map has to be removed. As depicted in Figure 7, removing a single filter in layer $i+1$ results in the removal of its corresponding feature maps and of the corresponding kernels in layer $i+2$. On the other hand, if we decide to zero out shared kernels, we perform the exact inverse operation, as once a shared kernel is removed from the network in layer $i+2$, the corresponding input feature map is now useless and can also be removed. As a result, the corresponding filter in layer $i+1$ can also be removed.

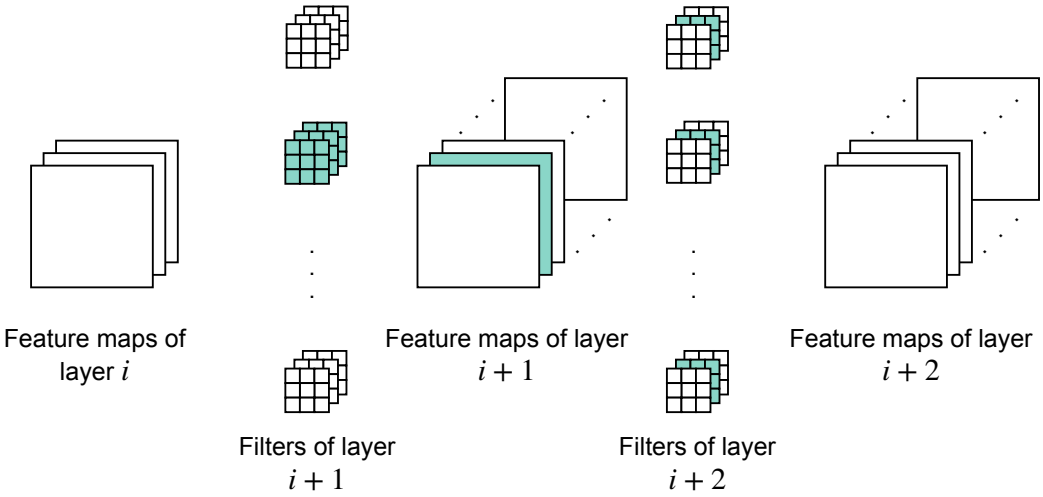


Figure 7. Filters containing zeroes can be removed from the model. By doing so, the corresponding feature maps, as well as their corresponding channel in the next layer, have to be removed.

As it removes parameters that have no impact on the computation of the result, the pruning is considered to be lossless, as it reduces the number of parameters and operation of the network, without altering its performance. To perform such an operation in FasterAI, the code required is expressed in Listing 9, with model being the model, sparsified according to the filter granularity beforehand.

Listing 9. Code required to prune a filter-sparse model.

```
pruner = Pruner()
pruned_model = pruner.prune_model(model)
```

Such a technique is currently only available on strictly feed-forward operations. Indeed, the implementation for operations containing skip connections is not straightforward, as there is no exclusive connection between a filter and its corresponding kernels.

3. Distill

FasterAI also brings knowledge distillation [24] capabilities to users with the help of its Distill module. Knowledge distillation methods are a set of techniques involving student–teacher-based training. In such a training, a large and performant model (the teacher) guides a small and less performant model (the student) in its learning process, as depicted in Figure 8. Knowledge distillation can generally be used to make a teacher provide information about its predictions, and a chosen loss is applied to encourage the student to replicate those predictions. The loss is applied on the respective logits of the teacher and student and is thus called the Logits loss (L_l). A teacher may also be used to provide information about intermediate computation states, e.g., activation maps. The loss responsible for incentivizing the student to replicate similar computation states is called

the feature loss (L_f). A total knowledge-distillation loss can be interpolated from those two losses and the classic training loss (L_c), e.g., cross entropy between student's predictions and data labels, with two interpolation parameters $\alpha[0,1]$ and $\beta[0,1]$, as

$$Loss = \beta(\alpha L_l + (1-\alpha)L_f) + (1-\beta)L_c$$

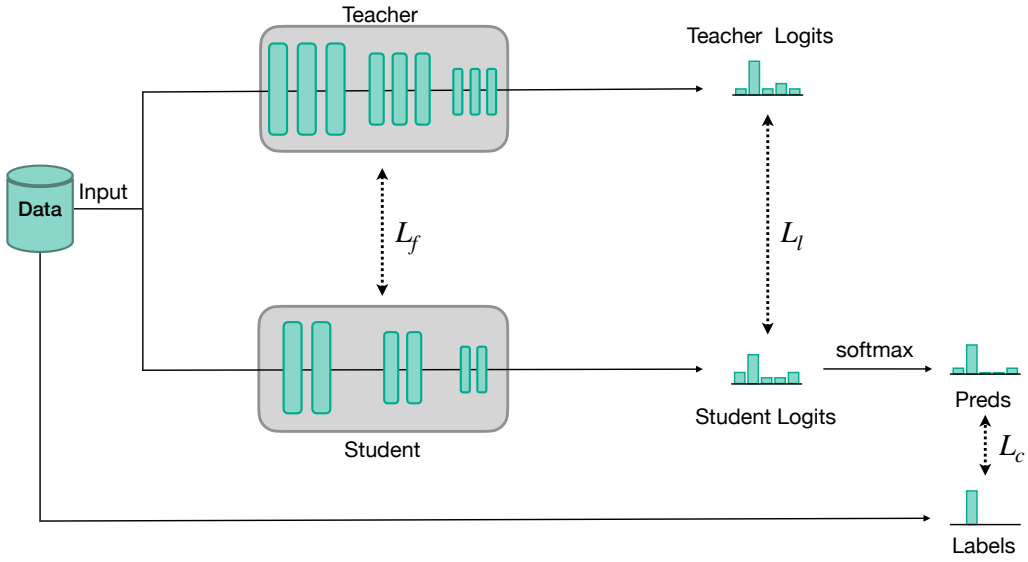


Figure 8. Illustration of the knowledge distillation process. Besides learning the labels from data, the student is also given some cues from the teacher, either from intermediate features, or from the teacher's predictions.

In FasterAI, this is managed by KnowledgeDistillationCallback, which offers Knowledge Distillation capabilities in a single line of code. As knowledge distillation is managed by another callback, it can be used in conjunction with SparsifyCallback, for even more flexibility for extreme compression or performing original experiments. The FasterAI usage for the KnowledgeDistillationCallback is given below in Listing 10, where layers_std and layers_tch are optional lists of layers, which will be used to compute the feature loss L_f if desired.

Listing 10. Code required to perform knowledge distillation in FasterAI.

```
kd_cb = KnowledgeDistillationCallback(tch, L_l, L_f, layers_std, layers_tch, α, β)
```

Knowledge distillation losses can be modified or created according to the user's needs. There are currently 3 logit losses and 4 feature losses available by default in FasterAI. We compare two of those losses in the same training conditions as previous experiments. In this scenario, the teacher model is a ResNet-34 model trained for 30 epochs from pretrained weights, and the student is a ResNet-18 model starting from random initialization. In particular, two distillation losses are compared with different interpolation values of β: (1) SoftTarget, the loss computed between the logits of the teacher and the student and (2) Attention, a loss computed from features extracted after each residual block of the teacher and the student. We report the results in Table 6. It can be observed that basing the knowledge distillation process on logits provides better results than attention. While SoftTarget compares the respective predictions of the teacher and the student, Attention holds a stronger hypothesis, that the layers used to compare are extracting the same information, which can make it harder to set up correctly.

Table 6. Results of applying knowledge distillation from a ResNet34 to a ResNet18 architecture for 4 different β interpolation values. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

SoftTarget				**Attention**			
0.3	0.5	0.7	0.9	0.3	0.5	0.7	0.9
83.32 ± 0.14	84.65 ± 0.31	84.74 ± 0.49	84.34 ± 0.17	81.56 ± 0.25	81.55 ± 0.41	81.64 ± 0.27	81.73 ± 0.67

4. Regularize

The regularize module of FasterAI concerns regularization techniques reducing the magnitude of weights in the network, according to a chosen granularity. This technique is often called weight decay when it concerns the granularity of weights. In practice, it adds a penalty term to the training loss. This penalty term acts as a regularization term, pushing the group of weights toward a value as small as possible during the optimization process. When the regularization is used to penalize weights according to their l_1-norm, it creates sparsity in the network. Eventually, this acts as a feature selection method, sparsifying some weights according to the desired granularity. However, as it is dependent on the optimization process, the sparsity level cannot be defined beforehand. It is nonetheless possible to control the importance of the penalty, to impose more or less sparsity in the final network, thanks to a penalty factor α. The final loss thus receives an extra term, adding the absolute value of weights for each layer l, according to the chosen granularity, as

$$Loss = L_c + \alpha \sum_l R(W_l)$$

with L_c as the classification loss, generally a cross-entropy computed between the predictions and the labels, and $R(W_l) = \frac{1}{G} \sum_g \sum_i |w_{g,i}|$ as the regularization term, G being the number of elements in each group. Such regularization can be applied in FasterAI by using the RegularizationCallback, according to a chosen granularity. This callback is presented in Listing 11.

Listing 11. Code required to perform group regularization in FasterAI.

```
reg_cb = RegularizationCallback(granularity, α)
```

We provide the results of the experiments conducted for different values of α in Table 7. As can be observed, a higher value of α leads to a degradation in accuracy, as too much penalty is being added to the loss value, making the optimization process put more emphasis on having small magnitude weights instead of an accurate network. Moreover, we can see that, as opposed to sparsifying, regularization performs better for more coarse granularities. This can be explained by the fact that the penalty value is dependent on the granularity structure, as the l_1-norm is averaged over the size of each block. This means that smaller structures will be penalized more, with the regularization term driving the loss value, thus giving more importance to the l_1-norm of weights than to the correct classification of data.

Table 7. Results of regularizing ResNet-18 with 4 different penalty strengths. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	1×10^{-7}	1×10^{-5}	1×10^{-3}	1×10^{-1}
weight	80.91 ± 0.44	79.45 ± 0.88	54.50 ± 0.94	23.91 ± 0.54
column	80.45 ± 0.29	81.02 ± 1.32	65.68 ± 1.32	25.91 ± 0.41
row	80.62 ± 1.31	79.39 ± 0.58	65.35 ± 0.49	25.67 ± 0.92
s-weight	80.11 ± 0.30	80.49 ± 0.65	80.67 ± 0.42	66.05 ± 0.59
channel	80.40 ± 0.49	81.56 ± 0.31	80.67 ± 0.11	63.02 ± 1.14
kernel	80.34± 0.14	81.29 ± 0.16	73.16 ± 0.14	27.81 ± 0.84
s-channel	80.35 ± 0.63	79.94 ± 0.59	80.31 ± 0.90	81.00 ± 1.22
s-column	80.09± 1.14	80.80 ± 0.31	81.46 ± 0.23	72.96 ± 0.49
s-row	80.71± 0.91	81.47 ± 0.30	80.29 ± 0.83	74.25 ± 0.21
v slice	80.46 ± 0.92	80.45 ± 0.52	80.43 ± 0.35	72.96 ± 0.84
h slice	80.73 ± 0.40	80.85 ± 0.85	80.98 ± 0.76	71.85 ± 0.63
s-v-slice	81.02 ± 0.74	81.31 ± 0.39	80.58 ± 0.74	80.56 ± 1.52
s-h-slice	80.40 ± 0.14	80.98 ± 0.89	80.12 ± 1.07	81.33± 1.92
s-kernel	81.42± 0.59	80.93± 0.57	81.66 ± 0.41	77.70 ± 0.84
filter	81.13 ± 0.43	81.15 ± 1.31	80.56 ± 0.47	76.48 ± 1.01

5. Misc

The last module of FasterAI is composed of compression techniques that do not fall into previous categories. In particular, two techniques are considered: (1) batch normalization folding; and (2) fully connected layer decomposition.

5.1. Batch Normalization Folding

The batch normalization layer is a normalization layer, usually placed between the computation layer and the non-linearity, and whose role is to normalize input data. This is performed by subtracting the mean and dividing by the standard deviation computed by a moving average on input data batches. Normalizing each incoming batch to a mean of 0 and standard deviation of 1 at each computation layer has been shown to greatly improve the training performance and to help to obtain a better-behaved optimization process. However, once the training has been completed, the moving statistics of each batch normalization layer are fixed. They can thus be incorporated into the computation layer preceding each of them. This can be achieved by re-expressing in a mathematically identical way, the weights and bias of the computation layer, taking the normalization effect into account. The output of a batch normalization layer y, is given by

$$y = \gamma \frac{z - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \tag{1}$$

with $\mu_{\mathcal{B}}$ and $\sigma_{\mathcal{B}}^2$ being the moving statistics computed on each input batch and γ and β, the learnable parameters of the batch normalization, and $z = W \times x + b$ the output of the previous computation layer, of weight W and bias b. We can thus re-express those weights and bias, accounting for the parameters of the batch normalization layer as

$$\begin{aligned} W_{\text{fold}} &= \gamma \cdot \frac{\mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \\ b_{\text{fold}} &= \gamma \cdot \frac{b - \mu}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} + \beta \end{aligned} \tag{2}$$

Once the weights and bias of the computation layer have been changed, the batch normalization layer can be considered useless and removed, slightly reducing the total amount of parameters and computation of the network. This operation is called batch normalization folding, and can be achieved in FasterAI by following Listing 12.

Listing 12. Code required to perform batch normalization folding in FasterAI.

```
bn = BN_Folder()
bn.fold(model)
```

Batch norm folding can also be considered as a lossless compression technique, as it does not affect the performance of the model. Indeed, as can be observed in Table 8, it allows to slightly decrease the parameter count, at no cost for accuracy.

Table 8. Results of performing batch normalization folding on ResNet-18. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Trained Model	**BN Folded Model**
Accuracy (%)	80.61 ± 0.42	80.61 ± 0.42
# Parameters	11,228,838	11,224,038

5.2. Fully Connected Layers Decomposition

Traditional neural network architectures such as VGG16 or AlexNet have fully connected layers accounting for up to 95% of their total amount of parameters, thus largely dominating the global storage footprint of such networks. An efficient way to reduce the contribution of the fully connected layer to the total amount of parameters is to decompose those layers using factorization techniques, such as truncated SVD. SVD decomposition allows to express a large weight matrix X into 3 smaller ones as:

$$X = U\Sigma V^*$$

with Σ being the diagonal matrix of singular values, ordered by importance. We can approximate the matrix X by selecting the leading $r \times r$ sub-block of Σ, and the corresponding r leading columns of U and V. The larger the value of r, the better the approximation will be.

We can choose the value r according to the desired compression rate. By then replacing the large weight matrix by its decomposition, we are able to reduce the number of parameters of the model. This can be achieved by applying Listing 13.

Listing 13. Code required to perform knowledge distillation in FasterAI.

```
fc = FC_Decomposer()
fc.decompose(model, pct_removed)
```

The pct_removed term corresponds to the percentage of singular values kept from the diagonal of the Σ matrix. The results of applying fully connected layer decomposition are reported in Table 9. As can be observed, high compression can be achieved before affecting performance. The compressed model can also be further fine-tuned to recover from the lost performance.

Table 9. Results of decomposing fully connected layers of ResNet-18 with 3 different compression levels. Mean and standard deviation of accuracy over 3 rounds are reported. The darker the shade of red, the further the accuracy is from the baseline.

	Trained Model	**pct_removed 25%**	**pct_removed 50%**	**pct_removed 75%**
Accuracy (%)	80.61 ± 0.42	80.84 ± 0.17	80.67 ± 0.61	77.24 ± 0.74
# FC Parameters	52,326	46,766	31,416	15,452

6. Conclusions and Future Development

In this paper, we detail the FasterAI library, which provides a lightweight framework enabling quick and diverse experiments on neural network compression techniques. More

particularly, we present the four modules along which the library is developed: (1) sparsify, concerning techniques introducing sparsity in neural networks; (2) distill, which concerns knowledge distillation techniques, helping a small model to reach a higher performance; (3) regularize, providing capabilities to perform grouped weight decay; and (4) misc, with other compression techniques such as batch normalization folding or fully connected layer decomposition. For each technique available in FasterAI, we provide extensive proof-of-concept experiments, performed with ResNet-18 trained on CALTECH-101, validating the different techniques available in the library, and demonstrating the range of parameters available by default.

More than just a compression library, we believe that the way FasterAI was built laid solid foundations to allow an easier implementation of novel compression techniques. Indeed, its unique granular approach to implementing compression techniques allows to seamlessly combine and customize them. Additionally, because it possesses many default options, it will help enthusiasts to apply compression techniques to their neural networks. Additionally, as demonstrated in the paper, because the implementation of novel techniques usually comes down to the writing of a single line of code, we hope that the library will help researchers in the field to create new compression techniques and to easily perform extensive experiments. We would like to continue developing FasterAI with the same philosophy in mind, striving for an increasingly flexible and convenient framework. We would also like to keep it up to date with new compression techniques, such as quantization [25] and conditional computation [26].

Author Contributions: Conceptualization, N.H.; methodology, N.H.; software, N.H.; validation, N.H.; formal analysis, N.H.; investigation, N.H.; writing—original draft preparation, N.H.; writing—review and editing, M.M.; supervision, M.M., B.G., M.P., T.Z.; funding acquisition, M.M., B.G., M.P., T.Z. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded in the context of a collaboration between the University of Mons (Belgium) and Télécom SudParis (France).

Acknowledgments: The authors would like to thank the research team members for their contributions to this work.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Blalock, D.W.; Ortiz, J.J.G.; Frankle, J.; Guttag, J.V. What is the State of Neural Network Pruning? In Proceedings of the Machine Learning and Systems, MLSys, Austin, TX, USA, 2–4 March 2020.
2. Hubens, N. Fasterai. 2020. Available online: https://github.com/nathanhubens/fasterai (accessed on 20 September 2022).
3. Liang, T.; Glossner, C.J.; Wang, L.; Shi, S. Pruning and Quantization for Deep Neural Network Acceleration: A Survey. *Neurocomputing* **2021**, *461*, 370–403. [CrossRef]
4. Hoefler, T.; Alistarh, D.; Ben-Nun, T.; Dryden, N.; Peste, A. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *J. Mach. Learn. Res.* **2021**, *22*, 10882–11005.
5. Wang, H.; Qin, C.; Bai, Y.; Zhang, Y.; Fu, Y. Recent Advances on Neural Network Pruning at Initialization. In Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, Vienna, Austria, 23–29 July 2022.
6. Facebook Research. Open LTH. 2019. Available online: https://github.com/facebookresearch/open_lth (accessed on 12 September 2022).
7. Changyu, C. Awesome-Deep-Neural-Network-Compression. 2019. Available online: https://github.com/csyhhu/Awesome-Deep-Neural-Network-Compression (accessed on 12 September 2022).
8. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of the Advances in Neural Information Processing Systems, NeurIPS, Vancouver, BC, Canada, 8–14 December 2019.
9. Kurtz, M.; Kopinsky, J.; Gelashvili, R.; Matveev, A.; Carr, J.; Goin, M.; Leiserson, W.; Moore, S.; Nell, B.; Shavit, N.; et al. Inducing and Exploiting Activation Sparsity for Fast Inference on Deep Neural Networks. In Proceedings of the International Conference on Machine Learning, ICML, Virtual, 13–18 July 2020.
10. Zmora, N.; Jacob, G.; Zlotnik, L.; Elharar, B.; Novik, G. Neural Network Distiller: A Python Package For DNN Compression Research. *arXiv* **2019**, arXiv:1910.12232.
11. Howard, J.; Gugger, S. Fastai: A Layered API for Deep Learning. *Information* **2020**, *11*, 108. [CrossRef]

12. Falcon, W.; The PyTorch Lightning Team. PyTorch Lightning. 2019. Available online: https://github.com/PyTorchLightning/pytorch-lightning (accessed on 21 August 2022).
13. He, K.; Zhang, X.; Ren, S.; Sun, J. Deep Residual Learning for Image Recognition. In Proceedings of the Conference on Computer Vision and Pattern Recognition, CVPR, Honolulu, HI, USA, 21–26 July 2016.
14. Li, F.-F., Fergus, R.; Perona, P. Learning generative visual models from few training examples: an incremental Bayesian approach tested on 101 object categories. In Proceedings of the Workshop on Generative-Model Based Vision in Conference of Computer Vision and Pattern Recognition, CVPR, Washington, DC, USA, 27 June–2 July 2007.
15. Kingma, D.P.; Ba, J. Adam: A Method for Stochastic Optimization. In Proceedings of the International Conference for Learning Representations, ICLR, Banff, AB, Canada, 14–16 April 2014.
16. Zhou, H.; Lan, J.; Liu, R.; Yosinski, J. Deconstructing Lottery Tickets: Zeros, Signs, and the Supermask. In Proceedings of the Neural Information Processing Systems, NeurIPS, Virtual, 8–14 December 2019.
17. Zhu, M.; Gupta, S. To Prune, or Not to Prune: Exploring the Efficacy of Pruning for Model Compression. In Proceedings of the International Conference on Learning Representations, ICLR, Vancouver, BC, Canada, 30 April–3 May 2018.
18. Hubens, N.; Mancas, M.; Gosselin, B.; Preda, M.; Zaharia, T.B. One-Cycle Pruning: Pruning ConvNets Under a Tight Training Budget. In Proceedings of the International Conference of Image Processing, ICIP, Hong Kong, China, 25–27 March 2022.
19. Frankle, J.; Dziugaite, G.K.; Roy, D.; Carbin, M. Pruning Neural Networks at Initialization: Why Are We Missing the Mark? In Proceedings of the International Conference on Learning Representations, ICLR, Virtual, 3–7 May 2021.
20. Han, S.; Pool, J.; Narang, S.; Mao, H.; Gong, E.; Tang, S.; Elsen, E.; Vajda, P.; Paluri, M.; Tran, J.; et al. DSD: Dense-Sparse-Dense Training for Deep Neural Networks. In Proceedings of the International Conference on Learning Representations, ICLR, Toulon, France, 24–26 April 2017.
21. Frankle, J.; Carbin, M. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In Proceedings of the International Conference on Learning Representations, ICLR, New Orleans, LA, USA, 6–9 May 2019.
22. Paul, M.; Chen, F.; Larsen, B.W.; Frankle, J.; Ganguli, S.; Dziugaite, G.K. Unmasking the Lottery Ticket Hypothesis: What's Encoded in a Winning Ticket's Mask? In *arXiv* **2022**, arXiv:2210.03044.
23. Frankle, J.; Dziugaite, G.K.; Roy, D.; Carbin, M. Linear Mode Connectivity and the Lottery Ticket Hypothesis. In Proceedings of the International Conference on Machine Learning, ICML, Virtual, 13–18 July 2020.
24. Hinton, G.; Vinyals, O.; Dean, J. Distilling the Knowledge in a Neural Network. In Proceedings of the Advances in Neural Information Processing Systems, NeurIPS, Montreal, QC, Canada, 7–12 December 2015.
25. Nagel, M.; Fournarakis, M.; Amjad, R.A.; Bondarenko, Y.; van Baalen, M.; Blankevoort, T. A White Paper on Neural Network Quantization. *arXiv* **2021**, arXiv:2106.08295.
26. Bengio, E.; Bacon, P.; Pineau, J.; Precup, D. Conditional Computation in Neural Networks for faster models. *arXiv* **2015**, arXiv:1511.06297.