UCLouvain

École polytechnique de Louvain

# Knowledge distillation analysis and evaluation for embedded applications on FPGAs

Author: **Jean-Martin Vlaeminck**
Supervisors: **Christophe De Vleeschouwer, Jean-Didier Legat**
Reader: **Martin Lefebvre**
Academic year 2020–2021
Master [120] in Computer Science and Engineering

**Abstract**

Deep learning has been instrumental in recent advances in artificial intelligence and computer vision, with countless applications. However, deep learning requires extremely large models to achieve its stellar performances, that require a lot of processing power both for training and when deployed in devices, and consume a large amount of energy. To reduce the power consumption in embedded applications, field-programmable gate arrays could be a possible solution, as they have a high energy efficiency while still maintaining superior performance than CPUs. But their limited size requires the use of much smaller networks, which are not as accurate.

Knowledge distillation is a technique of training deep networks where a large and accurate model (a teacher) is used to train a smaller but easier to deploy network (a student), by providing the latter with the outputs of the former as an additional objective. Knowledge distillation can typically train the student to a much higher accuracy than when trained alone, sometimes even surpassing its teacher.

This thesis aims to evaluate the use of knowledge distillation to train networks adapted for FPGAs, as well as the factors influencing its success. Our results show that distillation is indeed applicable with success.

We include a theoretical analysis of the impact of distillation on the loss and the gradients, and what this may mean concerning the mechanism of action of distillation.

We obtain that the best results are obtained with a higher temperature than usually used in knowledge distillation. We find that teachers whose training is stopped early are extremely effective, improving their student's accuracies by as much as 6% on CIFAR-100, and that distillation could potentially reduce the required training time.

# Acknowledgements

This thesis would not have been possible without the help and support of many people, whom I would like to thank here.

First, I would like to thank my thesis supervisors, Prof. Christophe De Vleeschouwer and Prof. Jean-Didier Legat, for their invaluable support, patience and advices during the realization of this thesis. They helped set a definite objective for this work and cleared many interrogations I had.

Next, I would like to thank my parents, who have always been here through my life, for their support, patience and reassurance during these two tough years. I will always love you.

I would like to thank my reader, Martin Lefebvre, as well as to Antoine Vanderschueren and Victor Joos de ter Beest for their help during this thesis.

I would like to address special thanks (and tanks) to Ulrich Winz, for his friendship and the countless fun moments throughout that helped me keep my sanity, or not.

Many thanks to my *cokotteurs*, for these two wonderful years of cohabitation despite this despicable virus.

I would like to thank many friends: Pauline, as well as Maxime, Martin, the Ingi Gang and last year's Parnas room residents.

In addition, I would like to thank the developers and maintainers of many of the tools I used during this thesis: PyTorch, TensorFlow, Vim, Tmux and Python, and many others that I have probably forgotten. The dedication to provide free and open tools for the community should be acknowledged.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the last decades, deep learning has virtually revolutionized artificial intelligence and computer vision. From face recognition systems to auto-correcting keyboards to self-driving cars and personal assistant, deep learning applications have been deployed all around us and sparked an intense new industry.

However, deep learning is hungry of processing power. Obtaining the high accuracy required for practical applications requires increasingly larger models, and these models require huge amounts of computational power to efficiently train them. Therefore, hardware accelerators for the training of deep neural networks are required, and graphical processing units (GPUs) have been traditionally used thanks to their large computational power, both for training and for inference.

But graphical processing units require a lot of energy to perform their computations, and are not adapted for embedded applications that need to operate at a low power level. For this reason, a lot of research has gone into developing power-efficient hardware accelerators for such applications. Field-programmable gate arrays (FPGAs) are one such platforms. They consist of a reprogrammable collection of hardware digital building blocks, that can be configured for a specific application with nearly as much energy efficiency as custom digital integrated circuits, but with a lower turn-around time, and they typically consume a lot less power than GPUs and CPUs.

Still, deploying deep learning models on FPGAs is a demanding task: one needs to program both the computational elements and the control units, there is a scarcity of ready-to-use framework to develop on the platform, and FPGAs still have a much lower amount of computational resources than GPUs. As such, development of FPGA implementations on FPGAs has typically focused on much smaller networks, which are much easier to implement, but conversely have a much lower accuracy than typical deep learning models.

Many techniques have been developed to produce small deep learning models, that can be efficiently deployed, from much larger and accurate models, while still retaining as much accuracy as possible: most notably, quantization and pruning. These techniques typically reduce the size of a initially large and accurate models to better fit an FPGA, and include mitigations to prevent the accuracy from deteriorating too much.

In recent years, another family of techniques has been developed, *knowledge distillation*. These techniques aim to train an already small networks, called the "student", by providing it hints from a much larger and accurate model, the "teacher", transferring the knowledge from the larger model to the smaller one, with the hope that the student reaches an accuracy close to its teacher. This idea can be traced back as far as 2006 with the paper on model compression from Bucilua and Caruana [5], but the most prevalent form of distillation, and the one we will analyze in this thesis, was introduced in 2015 in the seminal paper from Hinton, Vinyals and Dean [22]. The paper sparked a lot of research into the idea of transferring knowledge between two networks, from improving or devising new methods to implement this transfer [39], [62], to adaptations of the method to newer tasks [6], [41], as well as practical applications of knowledge distillation [2], [37].

Knowledge distillation, as proposed by [22], consists in scaling the outputs of both models by increasing a parameter called the *temperature*, in the softmax function present at the output of most classification models. This increased temperature should produce a less contrasted version of the output of the model, with more information preserved on the relative importance of the outputs. Then, the student is trained so that, in addition to minimizing its usual error rate, it also minimizes the difference between its outputs and the outputs of the teacher.

Knowledge distillation is still a relatively recent technique, and is not entirely well understood. Namely, there are many factors that can impact its success, and it also has shown surprising results, such as students beating their teachers. The fact that students can reach such a higher accuracy when trained with a teacher, that they cannot reach when trained alone, also raises some questions.

In this thesis, we want to investigate and possibly answer some of these questions. We evaluate the use of Hinton's knowledge distillation when training small networks better adapted for embedded applications, as well as the factors influencing its success.

We limit the scope of this thesis to the evaluation of knowledge distillation, as we consider other techniques like quantization and pruning as orthogonal to our main focus. We also only evaluated the form of knowledge distillation proposed in [22], as it is one of the easiest forms of knowledge distillation to implement in a current system.

Because deep neural network training is commonly performed by optimization algorithms like SGD that make extensive use of the gradient of a loss function, we first examined the impact of knowledge distillation on those gradients.

For the experiments designed to evaluate knowledge distillation, we use residual networks [17] and wide residual networks [63], which are some of the most widely used architectures in deep learning research. We use the CIFAR-100 dataset [28], a more challenging version of the commonly used CIFAR-10 dataset, but that requires much less computational power than the ImageNet dataset [9]. We use a relatively simple training setup, detailed in chapter 4, essentially with the goal of reproducing the original setup and results from [17] and [39]. Although not the state of the art, and with a less-than-optimal setup, this doesn't undermine the end goal of the experiments, which is to compare accuracies between different setups.

Then, we performed a series of experiments to determine the influence of each factor in knowledge distillation. CIFAR-100 being a less frequently used dataset, we first had to develop a set of teacher models for use in the subsequent experiments, with a range of accuracy between 70% and 80%.

A first set of experiment determined a set of good values for the temperature as well as the relative importance of distillation compared to the usual objective. This experiment showed that higher temperatures are almost always preferable, especially when force the training loss to focus on the distillation.

Finally, we tested how well students of various sizes can improve thanks to distillation, how the choice of teacher determines the success of the technique, and how the training of the teachers themselves impact their effectiveness. We noticed that the accuracy gain of the students mostly depends on the teacher used and on the student capacity, and it did not appear to depend on a particular matching of architectural characteristics.

The choice of the teacher turns out to be the most important factor for the success of distillation. But the choice depends much more on the way the teacher was trained, and less on its accuracy. Namely, we notice that excellent models on the dataset turn out to be mediocre teachers, while much smaller models with a lower accuracy and a much shorter training time can help the students reach up to 4%-6% more accuracy and even surpass the teacher.

## Structure of the thesis

This master thesis is composed of three chapters.

The first chapter presents background information that is necessary for the comprehension of the scope of this thesis, and is mainly split into two sections: deep learning, and field-programmable gate arrays. We first provide an overview of deep learning and the datasets commonly used in

the field. We then describe convolutional neural networks (CNN), the main workhorse of the last decade's successes in computer vision, as well as an overview of the various CNN architectures. In particular, residual networks are presented, a widely used architecture in the literature, and is the main architecture used in this thesis. Next, we provide an overview of the process and concepts used to train a deep neural network, with a focus on the objective function and the gradient descent optimization algorithm. We also review some techniques that can be used to improve the performance of deep neural networks, like their speed or their storage requirements, adapted for embedded applications. The second section presents field-programmable gate arrays, their usage for the deployment of deep learning models, and a review of implementations.

The second chapter presents the technique of knowledge distillation, the main topic of this thesis. After a first section that presents notations used in the chapter, we define knowledge distillation and present its theoretical motivations. We review other techniques that are related to knowledge distillation, either extensions of the method, adaptations, or complete alternatives. To gain more insights into knowledge distillation, we analyze its effect in a simple case, centered on its impact on the outputs of the model. Lastly, we analyze outputs of real-world models.

The third chapter evaluates the use of knowledge distillation in a practical situation, where we want to deploy small models better suited for embedded applications and deployment on FPGAs. For this, we determine factors that impact knowledge distillation, such as the temperature and the teachers, and reach some conclusions that are in line with other works.

Finally, we conclude with a reflection on what we have learned, and which future works could be performed to better understand knowledge distillation.

# Chapter 2

# Background

This chapter presents background information on deep learning, convolutional neural networks and their training, gives an overview of the techniques used to reduce their size and computational requirements, and presents accelerators for deep learning applications, including field-programmable gate arrays, as well as a review of the state of the art in the deployment of deep learning models on FPGAs.

## 2.1 Deep learning and convolutional neural networks

### 2.1.1 Deep learning

#### 2.1.1.1 Example tasks

Deep learning has been applied with success in numerous applications, ranging from face recognition to voice recognition and synthesis.

Each of these applications use a deep learning model to solve a particular task, with a particular goal:

- **Image classification**: What does this image represent? The model is expected to answer with a single output, or *class*, that best describes what the image means, contains or represent: this can go from objects like "chair" to actions like "dance".
- **Object detection**: Which objects are present in this image? Here, we expect the model to list each (pertinent) object in the image, along with an approximate location of the object within the image (ex: a bounding box), and assign a label or class to each object.
- **Image segmentation**: This is a refinement of object detection, where the image must be partitioned (segmented) into non-overlapping sets of pixels belonging to an object, or to the background of the image. This can be a preprocessing step in a processing pipeline before object detection, but it can also be a task by itself, e.g. to isolate pedestrians. Labeling the segments may also be part of the task.

There are also tasks related to audio processing (voice recognition), video processing, text processing (machine translation) etc.

#### 2.1.1.2 Datasets

Deep learning aims to train a network so that it performs well on a task, and as such requires a *dataset* of training samples.

A dataset is split in two parts:

- the *training set* contains samples aimed for training the model;

- the *test set* contains samples aimed to evaluate the final model on samples that is has never seen during its training, representative of the actual conditions the model will face in a concrete application.

Usually, the training set is further split into two sets:

- the *actual* training set, on which the network will be trained, i.e. the learnable parameters of the network will be adjusted to correctly predict the target output;
- a *validation set*, which is used to perform model selection (selecting a model out of several candidate models), hyperparameters tuning (selecting the best hyperparameters for the training process), or early stopping (stop the training before the expected end to prevent further overfitting). It is a kind of test set during the development part of the model, aimed to provide the model designer with a good estimation of the performance the model would have on the actual test set, without interacting with the test set. Indeed, as the hyperparameters and the models are chosen based on the performance on this validation set, the final model has also been "trained" on the validation set.

Some datasets provide an explicit validation set; this is especially the case for datasets used in competitions, as the test set is usually not available. If absent, a validation set can be constructed by splitting the provided training set into an actual training set and a validation set, commonly into a 90%/10% split respectively.

**Image classification datasets** There is a large number of datasets for image classification, as it is one of the oldest task of image processing. The datasets vary mainly in terms of image sizes, number of possible classes, and number of training samples.

**CIFAR-10 and CIFAR-100** The CIFAR-10 and CIFAR-100 datasets [28] are two datasets, developed in 2009 by Alex Krizhevsky, the future co-author of AlexNet [29]. Each of them consist of 50,000 $32 \times 32$ RGB tiny images with an associated label. For CIFAR-10, there are 10 such classes, while for CIFAR-100, there are 100 fine classes, grouped into 20 superclasses of 5 classes each. While the difficulty of CIFAR-10 is reduced, as the classes are vastly different, the CIFAR-100 dataset is more challenging due to the large number of classes, that require a much more fine detection, all while working with tiny $32 \times 32$ images.

**ImageNet** The ImageNet dataset[1] [9] is probably the most used deep learning dataset for high-performance networks, and serves as a common benchmark for researchers. It consists of nearly 15,000,000 images at various resolutions, split into more than 20,000 different classes. Most researchers however use a subset of the dataset used in the ImageNet Large Scale Visual Recognition Challenge [43]. The challenge was held every year from 2010 to 2017 and showcased the progress made in deep learning models over the years. The dataset used 1000 classes and over a million images, and performance was measured according to two metrics: either top-1 accuracy (the model's top guess predicts the correct label), or top-5 accuracy (the correct label is among the model's top 5 guesses). Although widely used, the large number of classes, high resolution of the images and large number of samples require huge models that take days to weeks to train, which is why it wasn't used in this thesis.

**Tiny-ImageNet** The Tiny-ImageNet dataset [2] is a downsampled version of a subset of the ImageNet dataset, developed in 2017 for a Stanford course. It is an interesting alternative to the ImageNet and CIFAR-100 datasets, having more classes but a larger resolution.

[1] `https://www.image-net.org`
[2] `https://www.kaggle.com/c/tiny-imagenet/data`

| Dataset | Image size | Color | $C$ | # Train | # Valid | # Test | SotA |
|---|---|---|---|---|---|---|---|
| MNIST | $28 \times 28$ | gray | 10 | 60,000 | - | 10,000 | 99.84% |
| CIFAR-10 | $32 \times 32$ | RGB | 10 | 50,000 | - | 10,000 | 99.37% |
| CIFAR-100 | $32 \times 32$ | RGB | 100 | 50,000 | - | 10,000 | 93.51% |
| ImageNet | variable | RGB | 1000 | $\approx$1 million | 50,000 | 100,000 | 88.5% |
| Tiny-ImageNet | $64 \times 64$ | RGB | 200 | 100,000 | 10,000 | 10,000 | 83.3% |

Table 2.1: Some datasets for image classification. $C$ is the number of classes; the three next columns are the size of the corresponding set, in number of samples; color can either be gray (1 input channel) or RGB (3 input channels).

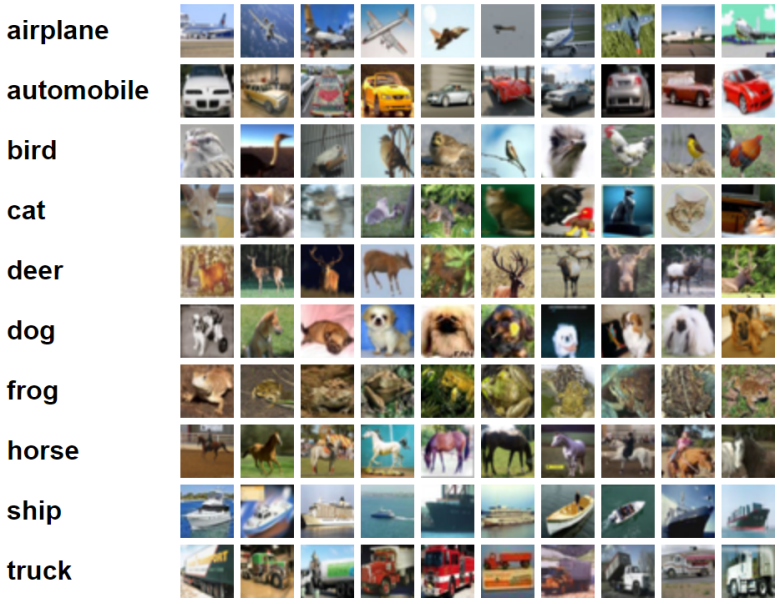



Figure 2.1: A sample of images from the CIFAR-10 dataset [28].

### 2.1.2 Convolutional neural networks

In the last few decades, the workhorses of computer vision have been **convolutional neural networks**, or CNN for short.

Convolutional neural networks are chiefly composed of a particular type of layer, the **convolutional layer**. Intermediate values in the network consist of a certain number of *channels* or **feature maps**[3], which are 2D spatial maps of values that represent the presence and intensity of a particular feature in the image (like edges, patterns, geometric forms, body parts or objects). Computing the output of a convolutional layer involves, for each output channel $c_o$, and for each location $(r, c)$ in this output feature map, moving a $K \times K$ *kernel* or *filter* over the image and multiplying it element-wise, across all input channels $c_i$:

$$\mathrm{Out}[c_o, r, c] = \sum_{c_i=1}^{C_i} \sum_{\Delta r=0}^{K-1} \sum_{\Delta c=0}^{K-1} \mathrm{W}[c_o, c_i, \Delta r, \Delta c] \cdot \mathrm{In}\left[c_i, r + \Delta r - \frac{K}{2}, c + \Delta c - \frac{K}{2}\right].$$

The $K \times K$ patch of image with which a kernel is multiplied is called its receptive field.

Convolutional networks also comprise other types of layers:

- *activation layers*, which consist of a simple *activation function*, whose aim is to introduce nonlinearities in the function learned by the network. The most commonly used is the rectified linear unit [38], or ReLU.
- *pooling layers*, which essentially work like convolutional layers, except that instead of computing a element-wise multiplication with a kernel over multiple input channels, they compute a statistic (average, max, min...) over the receptive field, with all channels being independent, and they reduce the size of the output feature map. This is typically used to achieve scale invariance. Most recent architecture ditch pooling layers in favor of computing the feature map size reduction directly by convolutional layers, by skipping every few row and column (the stride).
- *linear* or *fully connected* layers. They are typically found at the end of the network, and combine the various features extracted by the various channels into the output values of the network. They are not convolutional, instead taking all the inputs to produce a single value per output channel.

#### 2.1.2.1 Example architectures

**AlexNet** One of the first large-scale convolutional neural network is AlexNet, developed by Krizhevsky et al [29] in 2012. It contains five convolutional layers, three max-pooling layers after some of the convolutional layers, and three linear layers at the end, with ReLU activation throughout. This network, with 60 million parameters, achieved a top-5 test set error rate of 37.5% and a top-1 error rate of 17%, a 10% improvement over previous non-CNN-based techniques.

**VGG-Net** The 2nd-place winner of the ILSVRC 2014 is VGG-Net, developed by the Visual Geometry Group at Oxford University [48]. It features between 13 and 16 convolutional layers, and three fully-connected layers, with 5 max-pooling layers inserted every 2-4 conv. layers to halve the feature map size as we progress through the network. The major features of this architecture were: its use of kernels of size $3 \times 3$ for all conv. layers, and its simple rule of doubling the number of channels after (nearly) each of the max-pooling layers. The VGG-16 network, with 138 million parameters, achieved a top-5 test error rate of 24% and a top-1 error rate of 7.2%, and VGG-19 (144 million parameters) achieved a top-1 error rate of 7.1%, back in 2015.

[3]Both terms are used interchangeably; generally, an input image has three channels (red, green and blue), and the output of a layer consists of several channels of information, each channel consisting of a spatial map of values, each value indicating if a particular feature detected by this output channel is more or less present.

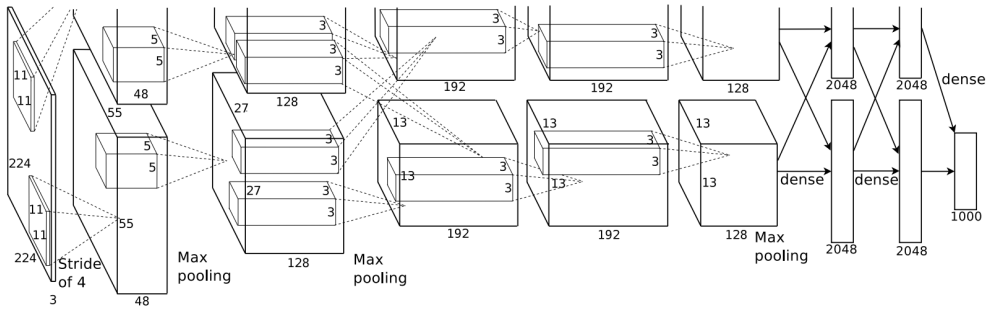



Figure 2.2: AlexNet architecture. The network was trained on two GPUs, with half of the channels of each layer per GPU, hence the horizontal split.

**GoogLeNet and the Inception family of architectures** The winner of the ILSVRC 2014 was GoogLeNet, an instance of a family of architectures named Inception [50]. This network features 22 layers of learnable parameters, for a total of 7 million parameters. A single network achieved 7.9% top-5 test set error rate, but a model consisting of 7 networks achieved 6.7% top-5 error rate.

This architecture used a bunch of techniques, later reused by other works, like modules (building blocks composed of multiple parallel layers with different configurations), bottlenecks (reducing the number of channels before entering a convolutional layer, to reduce the computations), and heaving data augmentation (modifying the input images to have a resistance against wide variations of the test images).

This architecture was the first of a family of architectures with the name "Inception" in it. Later versions, like Inception-V2 [27] and Inception-V3 [51] introduced factored convolutions (replace a convolution with a large kernel by a series of convolutions with a smaller kernel). Along with other tweaks to the training setup, Inception-V3 reached 18.8% top-1 and 4.2% top-5 test set error.

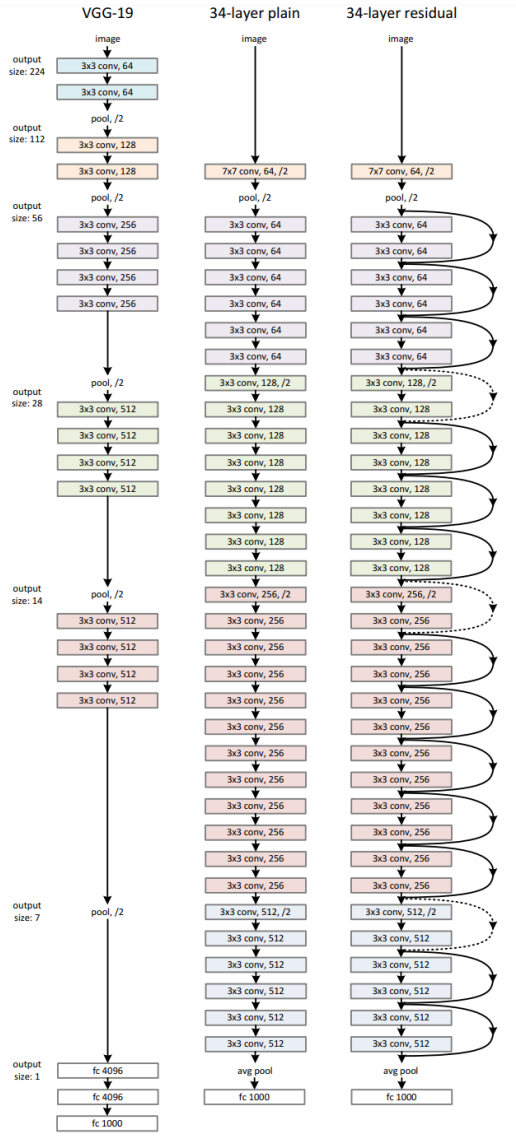
#### 2.1.2.2 ResNets

In 2015, residual networks [17] were proposed by He et al, and subsequently won the ILSVRC 2015. Residual networks (ResNets, or RN for even shorter) are structured like VGG networks, but with the addition of a *shortcut connection* every two layers (three for networks with bottleneck blocks); these blocks of layers with a shortcut path connecting its input and output is a *residual block*[4]. This shortcut connection helps the training by providing a mostly direct path along which gradients can flow, unobstructed by the weight layers that, especially at the beginning of the training, are unreliable in terms of the function they represent. ResNets however keep the same expressiveness as non-residual networks, and they can also learn easily to ignore parts of their layers by pushing their weights to zero, essentially becoming shallow networks. Indeed, ResNets were among the version networks to break the 30-layers barrier and gaining in accuracy while doing so; even extremely deep networks with more than a hundred of layers still managed to converge and provide accuracy improvements.

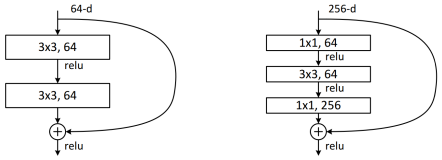
Residual and shortcut connections and residual networks have since become a common feature of deep learning models, being used in a lof of newer architectures [44], [49], [53], [60], [63].

There also exists a variant of residual networks called PreResNets [19], where the order of the layers inside the blocks is shifted to incorporate the output ReLU activation inside the block, leaving the path between blocks completely clean. As these activations are shifted just before the convolutional and batch normalization layers, they are said to *pre-activate the input.*

[4]The version with two layers of $3 \times 3$ convolutions is called a "basic block", to differentiate it from a "bottleneck block", where there are three convolutional layers of kernel size $1 \times 1$, $3 \times 3$ and $1 \times 1$, with less channels at the input and output of the $3 \times 3$ conv (hence the name).




(a) VGG (left and center) and ResNet (right) architectures.




(b) A residual block, with the curved shortcut path, and the residual path through the weight layers. On the left is a basic block; on the right is a bottleneck block.

Figure 2.3: ResNet architecture and its building block. The ResNet is composed of stacked residual blocks like the one in figure (b).

**Wide ResNets** Zagoruyko et al [63] introduced Wide ResNets (WRN), a variation of ResNets where instead of scaling the size of the network by increasing the *depth*, the network is scaled by increasing the *width*, or the number of channels/feature maps. Scaling networks by the width leads to more parallel computations, reduced latency and turns out to outperform the simple ResNets.

### 2.1.3 Training of deep neural networks

Neural networks need to be *trained*: we need to find values for the learnable parameters in the network so that the model achieves the desired level of accuracy. In this section, we describe the main concepts involved in this process. The interested reader can refer to any reference on deep learning; [12] is a good reference.

**Training as an optimization problem** The most common way of training a network is to reframe it as an optimization problem: we want to find values of the parameters $\Theta$ such that a particular function, the *loss function*[5], is minimized over the domain of possible parameters. Typically, the loss function is chosen so that its minimization coincides with the accuracy goal: the lower its value, the higher the accuracy.

**The cross-entropy loss** For classification tasks, a popular loss function is the *cross-entropy loss*. Given two probability distributions $P$ and $Q$ over the same set of events $\mathcal{E}$, the cross-entropy of the distribution $Q$ relative to the distribution $P$ is defined as

$$\text{cross-entropy}(P, Q) = -\mathrm{E}_P[\log Q] = -\sum_{e \in \mathcal{E}} P(e) \log Q(e). \tag{2.1}$$

The distribution $P$ is the reference distribution. In the context of machine learning, this will be the true probability distribution of the various labels for a particular sample from the dataset. The distribution $Q$ is then the distribution of the labels predicted by the model for this sample, and is directly dependent on the values of the parameters of the model.

Typically, a dataset will only contain a single label (the correct class $y$) for each sample $x$, so $P$ is effectively the distribution $P(e) = 1$ if $e = y$, and 0 otherwise. Then, noting $q$ the vector output of the model, and $q[i]$ the $i$-th component of this vector, the cross-entropy loss for the sample $x$ is usually expressed as

$$\text{cross-entropy}(\mathbb{1}_y, q) = -\log(q[y]) \tag{2.2}$$

where $\mathbb{1}_y$ denotes a distribution $P$ where $P(e = y) = 1$ but $P(e \neq y) = 0$. Our optimization problem for this particular sample $x$ then essentially consists in maximizing the $y$-th component of the output of the model so that the cross-entropy is minimal. If the model's prediction is then defined as the label with the largest corresponding output component, then by minimizing the loss we are effectively pushing the network to output the correct label.

An associated concept is the *Kullback-Leibler divergence*, noted $D_{\mathrm{KL}}(P||Q)$ and defined by

$$D_{\mathrm{KL}}(P||Q) = -\sum_{e \in \mathcal{E}} P(e) \log \left( \frac{Q(e)}{P(e)} \right). \tag{2.3}$$

This can be rewritten as

$$\begin{aligned} D_{\mathrm{KL}}(P||Q) &= -\sum_{e \in \mathcal{E}} \big( P(e) \log Q(e) - P(e) \log P(e) \big) \\ &= \text{cross-entropy}(P, Q) - \mathrm{H}(P) \end{aligned} \tag{2.4}$$

where $\mathrm{H}(P)$ is the Shannon entropy of the distribution $P$. The Kullback-Leibler divergence and the cross-entropy only differ by the entropy of the distribution of reference $P$. As this distribution is constant during training (as it is fixed by the dataset), we can equally minimize the KL divergence or the cross-entropy, whichever is easiest to compute.

[5] Also called *objective function*, though this term also includes functions that we want to maximize.

**Optimization by gradient descent** There are several ways to optimize an objective function, the most traditional being the *gradient descent*. We start from a initial state for the parameters $\Theta_0$, usually by assigning them from a random distribution. Then, we iteratively compute new values for the parameters $\Theta$. At each step $i$, with $\Theta_i$ being the current values of the parameters, we compute the loss function over all the samples in the dataset $\mathcal{X}$

$$\text{loss} = \sum_{x \in \mathcal{X}} f(\Theta_i, x)$$

and then, we compute the derivative of this loss with respect to all parameters in the model, the *gradient*

$$\nabla \text{loss} = \left[ \frac{\partial \, \text{loss}}{\partial \, \theta_i} \right]_{\theta_i \in \Theta_i}.$$

Next, we compute a new value for the parameters by computing

$$\theta_{i+1} = \theta_i - \epsilon \cdot \frac{\partial \, \text{loss}}{\partial \, \theta_i},$$

that is, we move the value of the parameter opposite the direction of increasing loss, and thus in the direction of decreasing loss. $\epsilon$ is a parameter of the learning process (an *hyperparameter*) called the *learning rate*: it determines the size of the step during the update. After a large number of such updates, the parameters of the model should be close to a *local minimum*, where the value of the loss function is locally minimal.

Computing the gradient of the loss w.r.t. each parameter in a network with multiple layers usually requires the *chain rule*: if the input to a layer is $x_i$ and its output is $x_{i+1} = f(x_i; \theta_i)$ with $\theta_i$ the parameters, then

$$\frac{\partial \, \text{loss}}{\partial \, \mathbf{x_i}} = \frac{\partial \, \text{loss}}{\partial \, \mathbf{x_{i+1}}} \cdot \frac{\partial \, (\mathbf{x_{i+1}})}{\partial \, (\mathbf{x_i})}$$
$$\frac{\partial \, \text{loss}}{\partial \, \theta_\mathbf{i}} = \frac{\partial \, \text{loss}}{\partial \, \mathbf{x_{i+1}}} \cdot \frac{\partial \, (\mathbf{x_{i+1}})}{\partial \, (\theta_\mathbf{i})}$$

where $\frac{\partial \, (\mathbf{f})}{\partial \, (\mathbf{x})}$ denotes the Jacobian matrix $J_{ij} = \frac{\partial \, f_i}{\partial \, x_j}$. Computing the gradients thus requires to go from the output to the inputs backward in the natural flow of the network, and is thus called *back-propagation* or the backward pass.

**Stochastic gradient descent** Gradient descent is an effective method of optimization, but it requires computing the loss function and the gradient over the whole training set at each step, which may require a prohibitive amount of computing resources. An alternative is to compute the loss function and its gradient over a subset of the training set, called a *batch* or *mini-batch* and denoted $\mathcal{B}$:

$$\text{loss} = \sum_{x \in \mathcal{B}} f(\Theta_i, x).$$

This method is called *mini-batch stochastic gradient descent* (SGD)[6]. At each step, the mini-batch is usually constructed by sampling a random subset of the training set. Its size usually depends on the computing resources available: a large size allows one to compute many forward and backward passes in parallel, but requires a larger amount of memory than a smaller batch size. Also, the distribution of samples with larger batch sizes are closer to the actual training set distribution, but this requires increasing the learning rate, which leads to convergence issues [20].

A notable variant of mini-batch SGD is mini-batch SGD with *momentum*: at each step, instead of updating the weights by using the gradient computing at the current step, we compute a running average of the past update vectors, so that the optimization keeps a form of momentum. This smooths the update vector and reduces the dependence of SGD on a single potential bad batch that would push the optimization in a completely different direction.

[6] Stochastic gradient descent only considers a single sample at each step.

**Learning rate schedule** The optimization process theoretically allows the network to converge to a local minimum. However, in practice, the value of the learning rate, combined with the randomness of training, causes the network to never actually converge and stabilize. To fix this, the learning rate is typically progressively reduced as the training advances, so that fluctuations in the network's weights and its accuracy are reduced at the end of training. The most basic learning rate schedule consist in dividing the learning rate by a factor at some epochs: for instance, we will be training for 200 epochs, and will divide the learning rate by 10 at epoch 100 and again at epoch 150. There exist other learning rate schedules, and there also exist variations of the optimizer where there is a per-weight or per-layer learning rate, allowing each part of the network to train at an appropriate speed.

#### 2.1.3.1 Regularization

Regularization is loosely defined as any technique whose aim is to improve generalization and decrease the test error [12], [65].

Deep neural networks are often over-parameterized: they contain enough parameters to fit to a randomly labeled dataset during training. As such, simply minimizing the training error may not lead to a model that generalizes well. To aid generalization, we can attempt to ensure that the training error is as close to the test error as possible: we need to "hurt" the training error so that it is larger.

**Batch normalization** Batch normalization [27] is a commonly used technique that has proven to improve the test accuracy of networks using it and speed up their training. It consists in *normalizing* the activations $x$ over a mini-batch $\mathcal{B}$, so as to ensure they keep the same overall value during the training.

In more details, if $x$ is a vector of $B$ values for a mini-batch, then we compute

$$\mu_{\mathcal{B}} = \frac{1}{B}\sum_{i=1}^{B} x_i$$
$$\sigma^2_{\mathcal{B}} = \frac{1}{B}\sum_{i=1}^{B}(x_i - \mu_{\mathcal{B}})^2$$
$$\hat{x_i} = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}}$$
$$x'_i = \gamma\hat{x_i} + \beta,$$

where $\gamma$ and $\beta$ are two learnable parameters, and $\epsilon$ is a small value added for numerical stability.

Batch normalization is usually applied as an additional layer in a neural network, and there is one such batch normalization layer per weight layer (convolutional or linear), usually applied just after. Virtually all CNNs use batch normalization layers, including ResNets.

Even though batch normalization is an effective way to improve deep neural networks, the reasons it works are still being debated, although a possible explanation is that it makes the optimization landscape smoother, leading to faster training [45].

**Weight decay or L2 norm** Weight decay is another commonly used technique to aid generalization. It consist in adding to the loss function an additional term that is the $l_2$ norm of all the parameters in the network:

$$\text{loss} = \text{loss}_{CE} + \text{wd} \cdot \sum_{\text{all parameters } \theta} \theta^2,$$

where wd is an hyperparameter that is usually called the weight decay.

Weight decay essentially prevents the parameters from becoming excessively large, as would happen in the case of an overfitting model, by sanctioning such an increase.

In practice, the value of the weight decay hyperparameter (from now on, "weight decay") is usually determined experimentally. A common value for the CIFAR dataset is 0.0001, although larger values are sometimes used.

The combination of weight decay with the use of some form of normalization (like batch normalization) causes the weights to stay in a much smaller range, and increasing the weight decay often has the same impact as increasing the effective learning rate, without its drawbacks [23].

### 2.1.4 Improving the performance of deep neural networks

Deep learning models are very accurate. But they are also very large and computationally intensive. Numerous works have thus attempted to reduce the size of the networks used in deep learning, or accelerate their computation.

Some works attempt to generate more efficient networks: this is the case of SqueezeNet [25], MobileNet [44], ShuffleNet [35], EfficientNet [53] and the related MnasNet [52].

Other works attempt to take a large and wide network, and remove seemingly redundant structures, channels or filters from it, a technique known as *pruning*. There is a large body of works on finding pruning schemes that maximize the resulting network's accuracy ([16], [30], [21], [33]) as well as works exploring its usefulness ([34], [61]).

Finally, other works focus on reducing the size of a model by reducing the size of its parameters: instead of computing with floating-point numbers, we can compute with fixed-point numbers, or even 4-bit or 2-bit integers: this is known as *quantization*. Several works have been performed on quantization, like binarization [8] or ternarization [10]. This has especially been of great use in FPGA implementations [58].

## 2.2 Deep learning on Field-Programmable Gate Arrays

Field programmable gate arrays (FPGAs) are a collection of digital building blocks and logic gates, that can be reconfigured to implement a particular hardware system, depending on the application. They can offer nearly the same features and performance as a custom-designed integrated circuit (application-specific IC, or ASIC), but can be reprogrammed to change the function of the system. They are especially useful when prototyping a hardware system, as ASIC development is extremely costly, or may be used in practical applications instead of ASICs, if the advantages of ASICs (reduced size, power consumption and improved performance) do not outweigh its cost.

In recent years, FPGAs have been considered to deploy deep neural networks. Advantages of FPGAs are mostly due to their lower cost compared to ASICs, while still beating most generic hardware accelerators on energy efficiency [59].

There are many works related to the optimization of convolution operations, as this is the most computation-intensive operation [36], [64]. Techniques used include loop unrolling, tiling, reordering, with the particular order and choice of operation determined by a design space exploration procedure that tries to optimize some target objective, like latency, throughput, energy efficiency or area utilization.

Venieris and Bouganis proposed several iterations of a system called fpgaConvNet [55]–[57], that can process modern architectures like ResNets and DenseNets [24].

FPGAs are also the platform of choice when it comes to deploy quantized neural networks, especially at very low bit width, as such networks can express their computations in terms of simple binary operations [32], [54], [58].

A more detailed review of FPGA implementations are provided in [15], [47], [59].

# Chapter 3

# Knowledge distillation

In this chapter, we present knowledge distillation, as proposed in [22].

We first present some notations used in the rest of the chapter.

Then, we formally define knowledge distillation and its motivation. We present the notion of temperature, and its relation with the standard cross-entropy loss used in most classification tasks. We also

We follow with a presentation of works related to knowledge distillation: works that preceded and inspired the technique, works that proposed adaptations or new techniques of knowledge distillation, and works that extended its reach to new use cases or new configurations.

To better understand the behavior of knowledge distillation, we perform an analysis on the value of the gradients and the losses, for a simple 2-classes classification task, to understand the impact of the various components of knowledge distillation.

Finally, we take at look at the outputs of actual teachers, to identify the real-world conditions in which knowledge distillation is active.

## 3.1 Notation

A neural network is typically composed of several layers, each performing an operation (see figure 3.1). In the case of a ResNet for instance, we have several blocks of a convolutional layer followed by a batch normalization layer and an activation function, over different feature map dimensions and with a certain number of channels per layer, each representing a particular feature, as well as shortcut connections between some layer outputs. At its end though, the network has to give an output that will be useful for the task at hand. For a classification task, where the network has to distinguish between $C$ different classes, the output will contain $C$ values, that we will call the *raw outputs* or *logits*, noted $z$. Usually, these values are obtained as the output of some linear layer at the very end of the network (layer $n$ in the figure). However, these values are usually not directly interpretable, and the final decision (predicted class) requires all the values (ex: take the class corresponding to the maximum value).

For the next sections, we need a clear notation to represent some concepts, like the components of a vector or a tensor, and a sample from a set (like a mini-batch). let $a$ be some quantity; for instance, it may be a sample (as a vector or a 3-channel image) (usually $x$), or the output of some layer (usually $a$ or $o$), or the output of the model (usually logits $z$ or softmax outputs $p$ or $q$).

- $a^{(i)}$ denotes that quantity for the $i$-th sample in some ordered set (typically the dataset, but it can also be the mini-batch). For instance, $x^{(i)}$ is the $i$-th sample, and $z^{(j)}$ is the output of the model for the $j$-th sample.
- $a_S$ denotes the quantity $a$ in some specific context. For instance, it could be the output of the teacher model $z_T$ VS the output of the student model $z_S$, or it could be the output of the $k$-th layer of the $S$tudent model $o_{S,k}$.

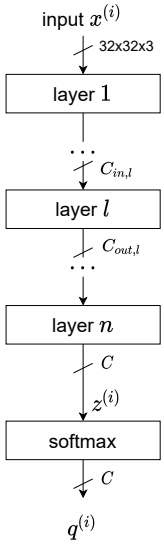



Figure 3.1: Typical structure of a convolutional neural network: from the input image $x^{(i)}$, the image is processed through several layers, each performing an operation (convolution, normalization, activation) and transforming the information into new representations. At the end, a linear layer produces an output vector $z$ of $C$ components, for the $C$ possible classes of the task at hand. An additional softmax layer converts them to pseudo-probabilities $q$, that saturate the output towards 1 (for the maximum) or 0 (the other components). Confusingly, the part closer to the input is usually called the *bottom* of the network, as they extract lower-level information from the image, while the part closer to the output is called the *top* of *head* of the network, as it contains high-level representations of the input.

- $a[k]$ denotes the $k$-th component of the vector quantity $a$. The components are numbered 1 to $N$, where $N$ is the size of the vector (ex: number of classes, number of channels etc). $a[i, j, k]$ could denote the component at entry $(i, j, k)$ in the tensor $a$ (typically used if talking about the feature map at some layer in the network).

For instance, $x^{(i)}$ may denote the $i$-th sample in the dataset, $o^{(i)}_{S,l,c}$ may denote the $c$-th channel of the feature map at the output of the $l$-th layer in a $S$tudent model, and $z^{(i)}_T[j]$ may denote the $j$-th component of the output of the $T$eacher model (the logit), when presented the $i$-th sample.

Finally, $\mathcal{X}$ denotes the dataset, $\mathcal{X}_B$ may denote a mini-batch of samples from the dataset, with a shortcut notation $\mathcal{B}$. An exponent like $\mathcal{X}^n$ denotes the set of all $n$-tuples whose components are taken from the set $\mathcal{X}$.

The most common way to extract interpretable information from the output is to apply a *softmax* layer on the output values, whose output are normalized so that they sum to 1. The softmax layer essentially generates "probabilities" for each possible class from the output of the model. We will note these softmax outputs as $q$ (or $p$ if they come from a reference model). The softmax function over a vector $a \in \mathbb{R}^C$ is defined as:

$$\text{softmax}\colon \mathbb{R}^C \mapsto \mathbb{R}^C \colon z \mapsto \text{softmax}(z)[i] = \frac{\exp(z[i])}{\sum_j \exp(z[j])} \quad \forall 1 \leq i \leq C. \tag{3.1}$$

One may wonder why there is a need to put a softmax layer at all, as it doesn't change the prediction from the model at all (the class corresponding to the maximum is the same in both cases). Actually, the softmax serves to compute that maximum when used with a loss function such as the cross-entropy: it has the effect of saturating to 1 the output corresponding to that maximum, and saturating to 0 the other outputs, in a continuous and differentiable way. In

addition, it gives a nice expression for the gradient of the loss w.r.t. the model outputs:

$$\frac{\partial \operatorname{loss}_{CE}^{(i)}}{\partial z[j]^{(i)}} = \operatorname{softmax}(z^{(i)})[j] - \delta_{y,j} = q(i)[j] - \delta_{y,j}, \tag{3.2}$$

where $y$ is the correct label for the sample $x^{(i)}$. Appendix C gives the details of the calculation of the gradients.

It is also interesting to have a look at the partial derivative of the softmax:

$$\frac{\partial \operatorname{softmax}(z)[i]}{\partial z[j]} = \operatorname{softmax}(z)[i] \cdot \big(\delta_{ij} - \operatorname{softmax}(z)[j]\big) \tag{3.3}$$

An

## 3.2 Knowledge distillation

Knowledge distillation (KD) was proposed in 2014 by Hinton, Vinyals and Dean [22], based on an idea that was already present as far back as 2006 [5].

Its idea, already explored by previous papers, is to use a large, cumbersome but very accurate model to train a smaller model, that can be more efficiently deployed in real applications (such as on-device computing, embedded applications, or even in datacenters with increased energy efficiency). The large model is designed as the *teacher model*, while the smaller model is the *student*. The motivation of knowledge distillation is that, if we can find a way to transfer the *knowledge*, that has been learned by the teacher during its training, to the student, then we can hope that the student could possibly behave like the teacher and reach a much higher accuracy than what it could have reached if trained alone, with only the same dataset. This knowledge is commonly assumed to be the function learned by the teacher, i.e. the output of the teacher when presented with a particular input, for all possible inputs.

Knowledge distillation is a different technique than *knowledge transfer*, which usually consists in taking a trained network, modifying parts of it (usually its output layers) and retraining it for a much shorter amount of time on a new task, by taking advantage of the knowledge already learned by the network during its first training; in that case, both backbone networks are practically the same.

There is a large body of works that shows that small and thin networks can perform as well as networks that are an order of magnitude larger and much more wide, provided they are trained correctly. Pruning schemes, for instance, often involve training a large and wide model, and then removing some of its filters or channels and recovering the lost accuracy ([33], [34]). Other works have shown that it is often significantly easier to train a very wide network than a similarly deep but thinner network ([11], [3], [31]). So, the apparently lower accuracy of deep but thin networks is likely due to our inability to train them to their potential, and not really an intrinsic property of these networks. In this context, knowledge distillation tries to directly train those small networks in a more informed way, instead of starting from much larger models that need to be pruned or converted in any way.

Hinton et al's knowledge distillation (HKD for short) works by using the softmax outputs of the teacher as an additional target objective for the student. Even though softmax outputs are not actual probabilities, they do have the same properties, and the softmax output of a teacher when presented with a sample can be considered as a form of probabilities that the teacher assigns to each label possibility for that sample.

The motivation of HKD is to consider the relations between these probabilities as a form of *dark knowledge*: namely, the relative value of the softmax reflects the relative similarity of the labels for the sample. For example, for an image of a car, it is expected that the pseudo-probability assigned by the teacher to the "car" class should be non-negligible and close to the pseudo-probability it assigns to classes similar to a car, like "bus" or "truck" (e.g., only an order of magnitude smaller), but it should be much greater than the pseudo-probability assigned to very different classes, like "carrot" (which should be many orders of magnitude smaller). In addition,

by giving the student an objective of mimicking the softmax outputs generated by the teacher, it can learn to generalize in the same way as the teacher, which should reduce its generalization gap.

However, the softmax outputs are usually extremely saturated, and it is not trivial to process the difference between such small quantities. To handle this problem, [22] proposes to soften the output of the softmax by dividing the raw outputs of the teacher and the student by a constant, called the *temperature* and noted $\tau$, which brings their values in a smaller range, thus reducing the saturation that softmax applies to the outputs. The temperature is an hyperparameter that should be chosen when training, and its value turns out to be capital to the effectiveness of knowledge distillation. Its name comes from the analogy with metallurgical processes like annealing, where an increased temperature is used to have a softer material to work with.

We define a version of the softmax function, with an additional parameter $\tau$, as:

$$\begin{aligned} \text{softmax}\colon \mathbb{R}^C \times \mathbb{R} \mapsto \mathbb{R}^C\colon z, \tau \mapsto \text{softmax}(z;\tau) = \text{softmax}\left(\frac{z}{\tau}\right) \\ \text{softmax}(z;\tau)[i] = \frac{\exp\left(\frac{z[i]}{\tau}\right)}{\sum_k \exp\left(\frac{z[k]}{\tau}\right)}. \end{aligned} \tag{3.4}$$

When referring to a network, whose output is $z$, we will refer to this softmax output as $q_\tau$.

The knowledge distillation process then consists in training the student on a *transfer set*, with an objective function that incites the student to produce similar softened pseudo-probabilities as the teacher. The distillation loss is defined as

$$\text{loss}_{HKD} = \tau^2 \,\text{cross-entropy}(q_{T,\tau}, q_{S,\tau}) = -\sum_{\text{class indices } i} q_{T,\tau}[i] \cdot \log\left(q_{S,\tau}[i]\right). \tag{3.5}$$

The cross-entropy loss ensures that the student learns to produce a similar output pseudo-distribution as the teacher.

However, it turns out that it is also beneficial for the student to have the usual cross-entropy loss (equation 2.2) as objective. So, the HKD loss is added to the usual cross-entropy loss by the way of a weight $\lambda_{HKD}$:

$$\text{loss} = \text{loss}_{CE} + \lambda_{HKD}\text{loss}_{HKD}. \tag{3.6}$$

The parameter $\lambda_{HKD}$ is another hyperparameter that has to be chosen when training. It allows specifying a greater focus on either the normal cross-entropy or on the knowledge distillation. Note that this definition is not the one from [22], which used a weighted average, but is the one from [42]. This doesn't change the learning process: indeed, what is important is the location of the optimum, and the relative importance of all the gradient components, and not the absolute value of the objective function.

**Temperature scaling of the HKD loss** There is one problem with the definition of the HKD loss in eq. 3.5: the magnitude of its gradient is dependent on the value of the temperature. Indeed, the gradient of the modified softmax function relative to its input scales as $\frac{1}{\tau}$ (for the non-saturated values):

$$\frac{\partial\, \text{loss}_{HKD}}{\partial\, z_S[j]} = \frac{1}{\tau}(q_{S,\tau}[j] - q_{T,\tau}[j]))$$

Thus, changing the temperature while experimenting with hyperparameters would also change the gradient, and thus the relative importance between the gradient from the cross-entropy loss and the gradient from knowledge distillation.

To remedy to this problem, we add a $\tau$ factor to the HKD loss expression:

$$\text{loss}_{HKD} = \tau \left( -\sum_{\text{class indices } i} q_{T,\tau}[i] \cdot \log\left(q_{S,\tau}[i]\right) \right). \tag{3.7}$$

However, there is still a problem: the gradient of this loss function still tends to zero as $\tau$ tends to infinity[1]:

$$\lim_{\tau\to\infty} \frac{\partial \operatorname{loss}_{HKD}}{\partial z_S[j]} = \frac{1}{\tau C}\big((z_S[j] - \overline{z_S}) - (z_T[j] - \overline{z_T})\big) = 0.$$

So, in order to have a nonzero gradient, it is necessary to multiply the HKD loss by an additional factor $\tau$:

$$\operatorname{loss}_{HKD} = \tau^2 \left( - \sum_{\text{class indices } i} q_{T,\tau}[i] \cdot \log\left(q_{S,\tau}[i]\right) \right). \tag{3.8}$$

This time, the loss function is complete.

This means that the HKD loss has an effective weight $\lambda_{HKD}\tau^2$ compared to the cross-entropy loss, and the gradient has a relative weight of $\lambda_{HKD}\tau$.

**The gradient of the HKD loss** The gradient of the HKD loss, with respect to the output of the student model $z_S$, is

$$\frac{\partial \operatorname{loss}_{HKD}}{\partial z_S[j]} = \tau\big(q_{S,\tau}[j] - q_{T,\tau}[j]\big). \tag{3.9}$$

As can be seen, it seems to scale with $\tau$, but as $s_{S,\tau}[j]$ for a non-predicted class $j$ scales as $\frac{1}{\tau}$, it stays relatively constant with varying temperature for outputs corresponding to non-predicted class.

**Limiting behavior of the gradient** With the $\tau^2$ factor in the loss function, we can compute the limit of the gradient for large $\tau$:

$$\lim_{\tau\to\infty} \frac{\partial \operatorname{loss}_{HKD}}{\partial z_S[j]} = \frac{1}{\tau C}\big((z_S[j] - \overline{z_S}) - (z_T[j] - \overline{z_T})\big) = 0. \tag{3.10}$$

It turns out that this is close to the gradient of the mean square error loss (MSE), considered by [4], [5]:

$$\operatorname{loss}_{MSE} = \frac{1}{2}\frac{1}{C}\sum_i (z_S[i] - z_T[i])^2 \tag{3.11}$$

$$\frac{\partial \operatorname{loss}_{MSE}}{\partial z_S[j]} = \frac{1}{C}(z_S[j] - z_T[j]), \tag{3.12}$$

the only difference being the $\overline{z_S}$ and $\overline{z_T}$ terms, representing the mean value of the vectors $z_S$ and $z_T$. As most neural network have some form of normalization at their layers, this is identical in practice. However, it means that with increasing temperature, the relative differences between the raw output values are increasingly uncovered by the saturating softmax and are increasingly pushed towards equal values.

**Values for the hyperparameters** The choice of the hyperparameters $\tau$ and $\lambda_{HKD}$ is of the most importance, as they impact the relative weight of knowledge distillation and standard training, as well as the range of logits that is softened. However, different works have used different values for them, as well as different definitions of the loss, and there seems to be no real consensus.

[22] used $\tau = 2 - -4$ and $\lambda_{HKD} = 0.5$[2], never being too explicit about the used values nor how other choices perform. [42] used $\tau = 3$, and a decaying $\lambda_{HKD}$ from $\frac{4}{3^2}$ down to $\frac{1}{3^1}$[3]. Attention transfer [62] uses $\tau = 4$ and "$\alpha = 0.9$" (probably $\lambda_{HKD} = 10$), the paper referencing an inexistent equation in [22]. [39] also uses $\tau = 4$ and "$\lambda_{HKD}$ to be 16 as in [22]" (again a non-existent

[1] The detailed computation can be found in the appendix, section C.3.1

[2] Or maybe 1, as the paper never explicitly writes down the loss function.

[3] The HKD loss does not include the $\tau^2$

variable, but likely to be the same as ours). And [7] uses $\tau = 4$, $\alpha = 0.9$ and claims them to be standard.

Overall, a typical choice for the temperature is $\tau = 4$, but the value for the weight $\lambda_{HKD}$ varies a lot between publications. During experiments, it will be necessary to determine which values lead to a good accuracy.

### 3.2.1 Interpretation of knowledge distillation

There have been much works and analyses done on knowledge distillation, with the goal of understanding why it works and how to make sure it works well.

The common interpretation is that knowledge distillation transfers the relations between the possible classes by transferring the relative value of the pseudo-probabilities: the probabilities for two classes that are similar would be close (e.g., 0.3 and 0.01 if the labels are very close to the predicted class, or 0.05 and 0.01 if they are less likely), while pseudo-probabilities for two classes that are dissimilar would be very different (e.g., 0.1 vs $1 \times 10^{-6}$). Notably, in front of a car, the teacher may assign a pseudo-probability of 0.6 to the "car" label, pseudo-probabilities of 0.1 and 0.15 to "truck" and "caravan", and a pseudo-probability of $1 \times 10^{-5}$ to "cat".

But there are several issues with this interpretation. First, we actually have no assurance that *this is what the network is doing*: it may very well put a pseudo-probability of 0.1 for the label "cat" and 0.15 for "carrot" in front of a car, as long as the correct label has a large enough pseudo-probability for the "car" label (e.g., 0.8). Indeed, we only tell the teacher about the *correct* label: it is never given any hint about which particular classes are similar, so it is free to assign probabilities to the incorrect classes as it wishes, as long as it minimizes the loss function. It would be interesting to verify if trained networks produce pseudo-probabilities that are indeed related depending on the class label similarities. This is especially important, as modern networks tend to be over-confident in their predictions, as incited by the cross-entropy loss, a phenomenon noticed in [14], which curiously mitigates the issue by increasing the temperature in the softmax function, like distillation does.

Another issue is that even if the pseudo-probabilities are indeed correct approximations of the relations between the class labels, then the fact that knowledge distillation transfers them at all is surprising. Knowledge distillation relies on this *distillation* aspect (i.e., increasing the temperature) to successfully transfer information. But the increase in temperature is usually limited to 2–4, thus the range of output differences is essentially widened by a factor of 2. It is unlikely to transfer information on the pseudo-probabilities for the classes with the lowest outputs, as their value is still dwarfed by that of the more likely classes. But classes with a non negligible output (e.g., the 2nd and 3rd predictions) are likely to benefit more. So, the temperature increase in KD should merely have an impact of bringing some labels to higher values so that they are considered by the HKD loss, but is unlikely to have an impact on lower probability labels.

But then, if knowledge distillation has increased the pseudo-probabilities of the 2nd, 3rd... labels, and thus also decreased the pseudo-probabilities of the predicted class, it means that the 1st and 2nd classes are increasingly closer pseudo-probabilities, and thus is reducing the sharp difference between the predicted class and the other ones. This could be potentially confusing for a student: why telling it that cars and vans are similar, if it can potentially not tell the difference initially, and risks mistaking them? However, in the special case where the teacher is making a wrong prediction, bringing alternative labels, among which the target label may be, can benefit the student, as with the aid of the true label (from the usual cross-entropy), it could more easily decide to ignore the teacher and lean on the true side.

Surprising too is the phenomenon where students manage to surpass their teachers, which has been reported in almost all explorations of knowledge distillation. If the student learns to imitate the output of its teacher, and turns out to beat it at the end, where does this increased accuracy come from?

Another interpretation would be that knowledge distillation is not transferring relations between pseudo-probabilities, but the actual softmax output assigned for the prediction, as a way to inform the student of its certainty and uncertainty about this sample. Larger values of the softmax would mean a higher confidence in the sample, and would essentially repeat the

normal cross-entropy loss. However, small values of the softmax for the predicted class could lead the student to avoid become overconfident in its prediction for this sample, and would reduce the natural tendency of the cross-entropy to oversaturate the output for the target class. A non-confident teacher for a particular sample would mean that the sample is difficult to classify, and instead of attempting to classify it (at the cost of potential misclassifications of other samples), the weighted loss would allow the student to position itself between the low certainty of the teacher and the high certainty encouraged by the normal cross-entropy loss, preventing it from overfitting this particular sample.

## 3.3 Other forms of knowledge distillation

Even though the knowledge distillation technique proposed by [22] is one of the most known and used technique to train a network with the knowledge of another, it is not the only technique. Many researchers have proposed various modifications to the technique since [22], and other techniques existed before its presentation.

In this section, we review other forms of knowledge distillation.

### 3.3.1 Works before HKD

#### 3.3.1.1 Direct matching of model outputs (*Model compression*, [5])

In 2006, Bucilă and Caruana [5] proposed to use a teacher-student framework (event though they didn't call it that way) to compress an ensemble of models into a single, smaller model. The method was applied to binary classification problems only; the target output was simply 0 or 1, and the models output a value in a continuum between 0 and 1. The student model was an artificial neural network, and was trained to mimic the output of the ensemble by minimizing the RMSE between the two:

$$\text{loss} = \sqrt{\frac{1}{\#\text{samples}} \sum_{x \in \mathcal{X}} (z_T(x) - z_S(x))}.$$

The transfer was performed on artificial data (the algorithm used to generate this data being the actual subject of the paper), hence the loss can't make use of the true label of the sample.

The paper demonstrated that artificial neural networks could perform on-par and even surpass their ensemble teacher, even though they couldn't reach the level of the ensemble if trained alone.

#### 3.3.1.2 Mimicking logits with an L2 loss (*Do deep nets really need to be deep?*)

In 2014, Ba and Caruana [4] used the teacher-student framework to show that shallow networks could be trained to have the same accuracy as deeper networks with the same number of parameters, if they are trained with the help of the deeper network.

The student is trained by minimizing the loss

$$\text{loss} = \frac{1}{2T} \sum_{x^{(t)} \in \mathcal{X}} \left\| z_S^{(t)} - z_T^{(t)} \right\|_2^2.$$

Samples are drawn from an unlabeled dataset, and the true labels are not involved during the transfer.

They also observed that shallow networks couldn't learn to have the same accuracy as deeper models of the same size, except when they have access to that deeper model. So, they suggested that improving the accuracy of the shallow networks was possibly a matter of using better algorithms for training them.

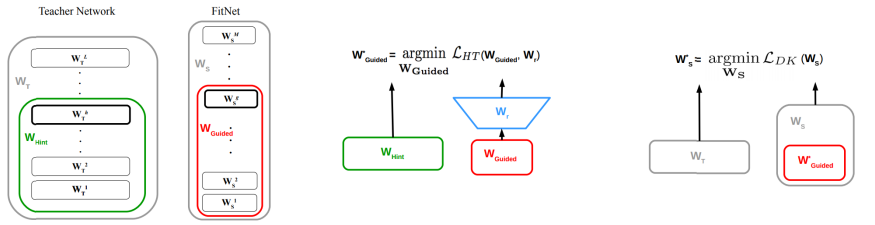



Figure 3.2: Working of FitNets. A part of the student is first pre-trained to mimic the output of a hidden layer of the teacher. Then, The full student is trained with knowledge distillation from the full teacher.

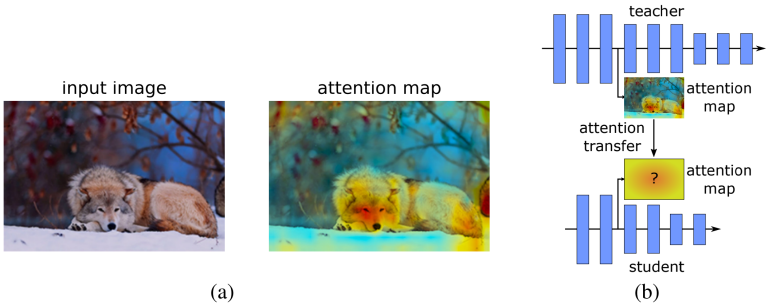



Figure 3.3: Attention transfer: the student network is trained with the goal of mimicking the attention map of a larger teacher model. Such an attention map can be defined in many ways.

### 3.3.2 Other forms of knowledge distillation after HKD

#### 3.3.2.1 Extracting mid-network information to initialize the student (*FitNets*)

Proposed end 2014 by Romero et al [42], FitNets, an extension of HKD, adds another component to the loss function in the form of hint-based training: giving the student an access to a hidden layer from the teacher (see figure 3.2). This hidden layer is typically at the half of each model, and is used to initialize a part of the student (typically the half too), by enforcing the student to mimic the output of the teacher's hidden layer. The similarity between the two layers is determined by an $l_2$ loss on the difference. As the student may not have the same number of channels at its extracted layer than the number of channels of the teacher's hidden layer, a linear regressor layer, $W_r$, with the correct output shape, is added on top of the student and trained with it; it will be discarded after the pre-training. After this pre-training, the training mainly follows that of [22], with a temperature of 3 and a weight $\lambda_{HKD}$ of 4 that is progressively decayed during the training.

#### 3.3.2.2 Transferring information from intermediate layers: attention transfer [62]

Hinton's knowledge distillation only transfers knowledge between the outputs of the two networks. However, valuable information may be contained in the intermediate feature maps of the teacher, as they contain a representation of what is contained in the image. Zagoruyko et al. [62] propose to *transfer attention*: where exactly in the feature map is the teacher looking at when processing an image (figure 3.3). For this, they define attention maps, based on the output of the models or on the gradients over these feature map. Then, the student is trained with an additional loss that encourages it to have similar attention maps to that of its teacher.

#### 3.3.2.3 Higher-order knowledge distillation (*Relational knowledge distillation*)

A common view of Hinton's knowledge distillation is that it gives to the student some information about the relationship between the different classes for a particular sample, such as how close or how different two classes are.

Park et al [39] propose an extension of this principle by transferring structural relations between outputs of multiple samples, instead of relations between the components of the output of a single sample. As told by [39], this closely relates to concepts of structural linguistics as proposed by Saussure, where the meaning of a sign depends on its relations with other signs in the system [46]. This *relational knowledge distillation* works by drawing $n$ distinct samples from the dataset, compute their outputs by the two networks, and minimize the following loss:

$$\text{loss}_{\text{RKD}} = \sum_{(x^{(1)},\dots,x^{(n)})\in\mathcal{X}^n} \ell\big(\psi(z_T^{(1)},\dots,z_T^{(n)}),\psi(z_S^{(1)},\dots,z_S^{(n)})\big), \tag{3.13}$$

where $\phi$ is a "potential" function, that extracts the relations between the samples, often by summarizing the relation between the $n$ samples. $\ell$ is an appropriate loss function used to measure the difference between the two potentials.

Park et al propose two concrete instanciation of RKD: the first one sets $n = 2$ and uses the Euclidean distance between the outputs for each of the samples. The second one sets $n = 3$ and transfers a relation between three samples in the form of the angle formed by the three samples in the output representation space.

The relational KD losses can be combined with the HKD loss and the task-specific loss, as an extension of HKD with additional weighted loss terms in the equation.

#### 3.3.2.4 Layer-wise progressive mimicking (*Go wide then narrow*, [67])

This form of "knowledge distillation" is a method to initialize a "student" from a previously-trained "teacher", before a fine-tuning step of classical training is performed. It is essentially an extension of FitNets, where the pre-training is applied many times with increasingly larger versions of the student.

We first divide the layers of the teacher and the student networks into $n$ "blocks"[4], or subsets of contiguous layers (see figure 3.4). Starting from the lower layers, a subset $S_i$ of the layers of the student are trained with a corresponding subset $T_i$ of the layers of the teacher, so that the difference between the outputs of the networks up to the currently considered blocks is minimized, when samples from the training dataset are shown to the inputs. During training, previously-trained subsets of the student are freezed. Similarly to FitNets, regressor layers are used to match the output shapes; however, these linear layers will be merged into an adjacent linear layer after this training. To measure the difference, they used the Kullback-Leilbler divergence, although they mentioned the mean-square error can also be used.

### 3.3.3 Knowledge distillation adaptations

Other works have also explored the use of knowledge distillation in tasks and situations where its direct application is not possible.

#### 3.3.3.1 Knowledge distillation with quantization

Even though quantization and knowledge distillation are techniques that are essentially orthogonal, their combination is not, as the quantization error needs to be taken into account, and there is a choice of what counts as the teacher – a regular full-precision large network, the quantized version of the large network, or the full-precision small network – and which version of the student is trained – the full-precision one, or the quantized one.

Polino et al. [41] investigated the use of HKD with quantization, with the distillation and quantization steps applied in a few different orders and on different networks.

[4] These blocks have no relationship with the notion of blocks in the context of residual networks.

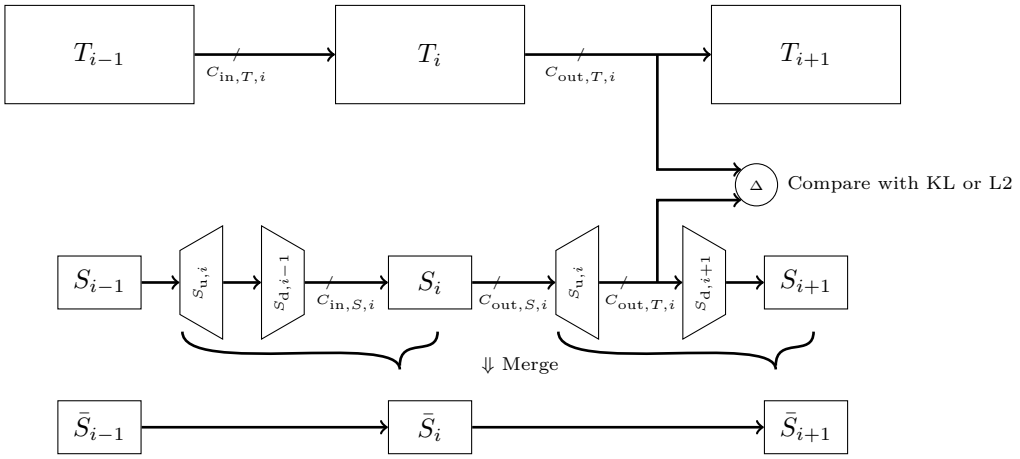



Figure 3.4: Go Wide Then Narrow: the teacher network is split into $n$ blocks $T_i$. Each output of the $i$-th student block $S_i$ has the same feature map size as the $i$-th teacher block $T_i$, but possibly a different number of channels. Then, we successively train the block $S_i$ of the student, as well as the linear adaptation layers $S_{\text{d},i}$ and $S_{\text{u},i}$, so that the output of the sub-student network $\{S_1, \ldots, S_i\}$ (with all the adaptation layers) mimics the output of the teacher network at the same level. The previous blocks and adaptation layers (everything before $S_{\text{u},i-1}$) are fixed during the training of the block $S_i$. At the end of this process, the adaptation layers are merged in an adjacent linear layer of one of the block (usually the block after it).

#### 3.3.3.2 Knowledge distillation on object detection tasks

In object detection tasks, the model must produce a bounding box that contains the object for each object it recognizes, as well as its label, instead of only a single label for the whole image, and must distinguish interesting objects from the background of the image. This causes the direct application of the HKD loss impractical.

Chen et al. [6] propose several modifications to the technique so that it can be used for object detection tasks, and performs several experiments on it. Some of the modifications include:

- Increased weighting of the background class VS the other classes in the knowledge distillation loss, for each object that is recognized and located by the network;
- Use of the teacher's regression output (the part that predicts the bounding box) as an upper bound for the student's regression output, ignoring the teacher when the student surpasses it;
- Use of hint learning, like FitNets [42].

## 3.4 Analyzing the effect of knowledge distillation on training

In the last section, we introduced knowledge distillation from a theoretical point of view. In this section, we want to analyze its workings by looking at its impact on gradients during training. Indeed, gradients are at the center of the optimization algorithm used during training, they are used to compute the update applied to parameters at each step. Analyzing the value and behavior of the gradients allows us to see how the knowledge distillation gradients will interact with the other gradients.

As visualizing gradients for a task with a large number of classes is essentially impossible, we settle on a simpler case: a binary classification task ($C = 2$). Such tasks are typically used to determine the presence/absence of an object. In that case, we only have 5 parameters:

- the two components of the logit of the student, $z_S[1]$ and $z_S[2]$;
- the two components of the logit of the teacher, $z_T[1]$ and $z_T[2]$;
- the temperature hyperparameter $\tau$.

Of these, only $z_S[0]$ and $z_S[1]$ can be learned, but the loss itself depends on the values of each of the 5 parameters.

In the following, we consider w.l.o.g. that the target class is the class 1.

**Temperature scaling and interpretation** We defined the HKD loss with a $\tau^2$ factor in front of the cross-entropy (eq. 3.8). This factor serves to guarantee that the gradients w.r.t. the non-predicted outputs stay constant when the temperature hyperparameter is varied. This scaling is well-motivated from a theoretical point of view, but is less motivated from a practical point of view. Indeed, when the temperature increases, so do the gradients of the HKD loss, and if we use a constant $\lambda_{HKD}$, we have that the contribution of the HKD loss starts to thwart the standard cross-entropy loss. Also, as the value of $\lambda_{HKD}$ can be chosen along with the temperature when performing the distillation, this square-of-the-temperature scaling could have been done by scaling $\lambda_{HKD}$ with the square of the temperature, without a factor of $\tau^2$; and the current scaling can also be compensated by a scaling of $\lambda_{HKD}$ with the inverse of the square of the temperature.

For the purpose of studying the losses, and their gradients, we introduce a generalization of the temperature scaling by defining the HKD loss as

$$\text{loss}_{HKD} = \tau^{\alpha} \left( - \sum_{\text{class indices } i} q_T^{\tau}[i] \cdot \log \left( q_S^{\tau}[i] \right) \right) \tag{3.14}$$

and its gradient by

$$\frac{\partial \, \text{loss}_{HKD}}{\partial \, z_S[j]} = \tau^{\alpha - 1} \big( \text{softmax}(z_S; \tau)[j] - \text{softmax}(z_T; \tau)[j] \big). \tag{3.15}$$

$\alpha = 2$ corresponds to the original scaling. $\alpha = 1$ corresponds to a "linear" scaling, that keeps the same magnitude for the gradient. $\alpha = 0$ corresponds to no temperature scaling: the $\lambda_{HKD}$ constant is the only parameter influencing the relative contribution of the two losses (beyond the actual values of the loss, that may still depend on the temperature).

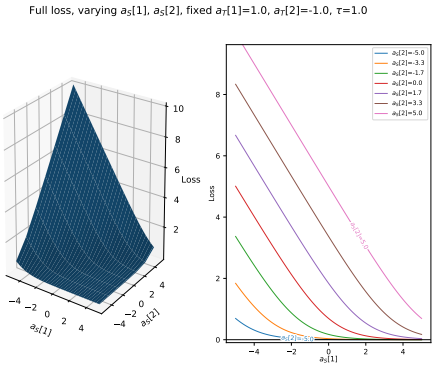
### 3.4.1 Cross-entropy loss

In this simplified case, the loss function is represented in figure 3.5. As we can see, the loss function strongly pushes the logits towards the area where $z_S[1] > z_S[2]$, so as to predict the first class with high "probabilities", but has a much lower gradient once this condition is satisfied.

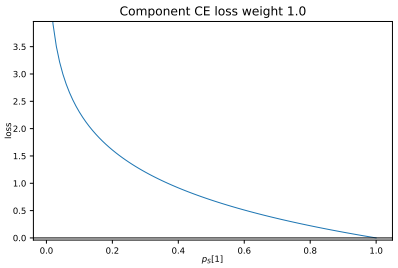
The logarithmic nature of the cross-entropy loss is visible in figure 3.5b, where we directly use the class "probability" as input. On figure 3.5a, we can see that when $|z_S[1] - z_S[2]| \gg 0$, the loss function is relatively linear, due to the logarithm and the exponential behavior of the softmax function at its limits.

The gradient of the loss has also a predictable form. With respect to the logits, it is just the softmax function (shifted one unit down due to the $\delta_{y_{\text{target}},j}$), and it saturates to $-1$ when $z_S[1]$ is too small. With respect to the class probability, it is just the negative of the $1/x$ function.

Note that the graphs on the second line give the gradient of the function with respect to the second component of the logit/class probabilities vector, but *depending on the value of the first component*, not of the second component. For $q_S$, we have that $q_S[2] = 1 - q_S[1]$, so it is just the negative of the above gradient. For $z_S$, one can prove that it is also the negative of the gradient w.r.t. the first component.

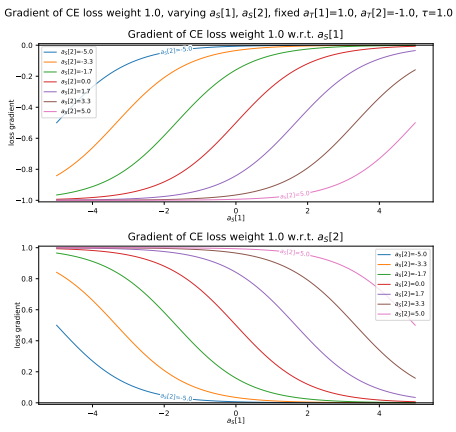



(a) Cross-entropy loss as a function of student logits. $a_S[1]$ is the component $z_S[1]$, the same goes for $a_S[2]$ and $z_S[2]$.

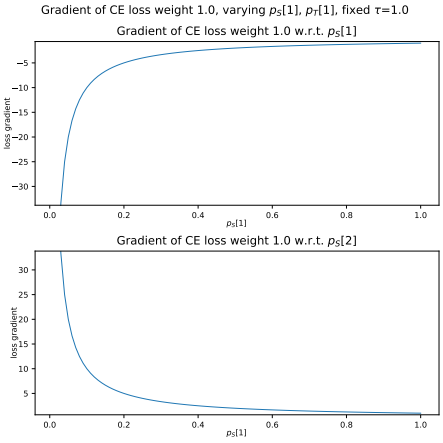



(b) Cross-entropy loss, with the first class "probability" per the student as input.

Figure 3.5: Cross-entropy loss function. $z_T$ and $\tau$ do not have an influence on this loss.




(a) Gradient of the loss relative to the student logits as inputs.




(b) Gradient of the loss relative to the student class "probabilities".

Figure 3.6: Gradient of the cross-entropy loss function. $z_T$ and $\tau$ do not have an influence on this loss.

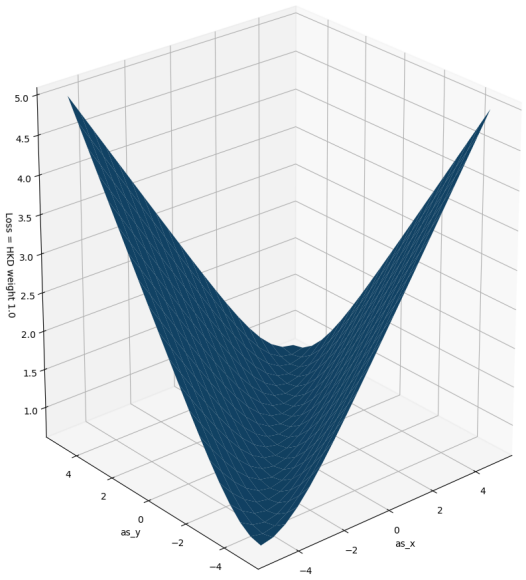



Figure 3.7: Knowledge distillation loss, $z_T = [1, 1]$, $\tau = 1$.

### 3.4.2 Knowledge distillation loss

In the case where $z_T = [1, 1]$ and $\tau = 1$, the loss function is represented in figure 3.7. In this case, the loss function strongly pushes the logits towards the area where $z_S[1] = z_S[2]$ (as the teacher's logits are also such that $z_T[1] = z_T[2]$), with no preference in the actual value of $z_S[1]$.

Another interesting observation is that the gradients are constant in the area away from the minimum valley, with the greatest gradient being orthogonal to the locus of minima.

**Impact of the teacher's relative confidence** During its training, a teacher will acquire knowledge of the task in order to predict with high confidence the correct target. Knowledge distillation assumes that the teacher will also learn a form of "meta" knowledge, or *dark* knowledge, in the relative importance of class probabilities with respect to each other. In our $C = 2$ case, this means that the teacher should learn about the relative importance of the two classes 1 and 2.

Given $z_T[1]$ and $z_T[2]$, the valley of minima has the equation:

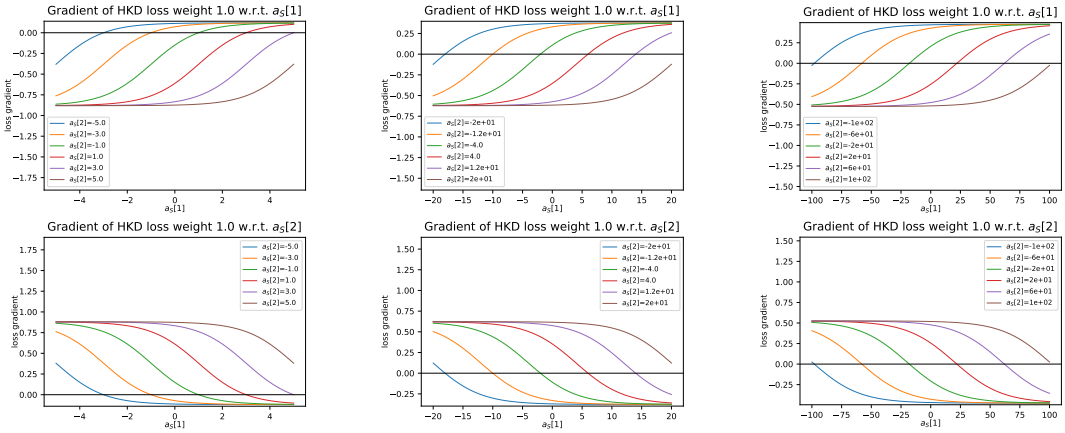
$$z_S[2] - z_S[1] = z_T[2] - z_T[1].$$

A first observation is that the value of the temperature $\tau$ doesn't influence this equation.

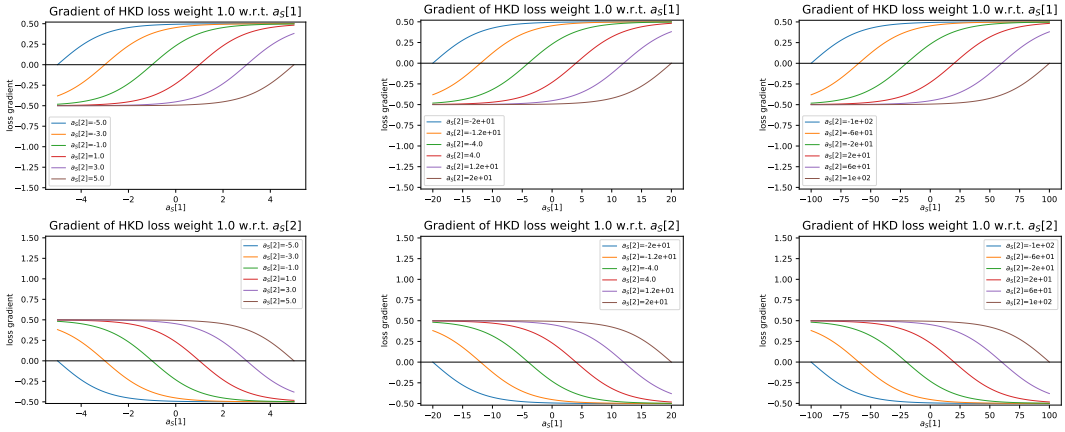
**Graphical analysis of the loss and gradients** Figure 3.8 shows the gradient of the HKD loss, with respect to the student logits, depending on $z_S[1]$ and $z_S[2]$, for various teachers and at various temperature. We can make the following observations:

- The size of the near-linear region[5]of the gradient increases linearly with the temperature $\tau$. Indeed, when $\tau$ increases, the argument of the softmax function, $\frac{z_S[i]}{\tau}$, decreases, so in order to keep the size of the "linear" region unchanged, the range of $z_S[i]$ has to be increased.
- Taking the limit where $\tau \to \infty$, this linear region becomes essentially infinite, which is explained by eq. (C.9).
- If the teacher is relatively certain ($z_T = (1, -1)$, top rows), the gradient will be much more negative on the region $z_S[1] \ll z_S[2]$ (corresponding to a bad prediction, or a mild

[5]One can loosely define a linear region of a function $f(x)$ as a range of values for $x$ where the function can be approximated by an affine function $\hat{f}(x) = a(x - x_0) + y_0$ to a satisfiable extent.

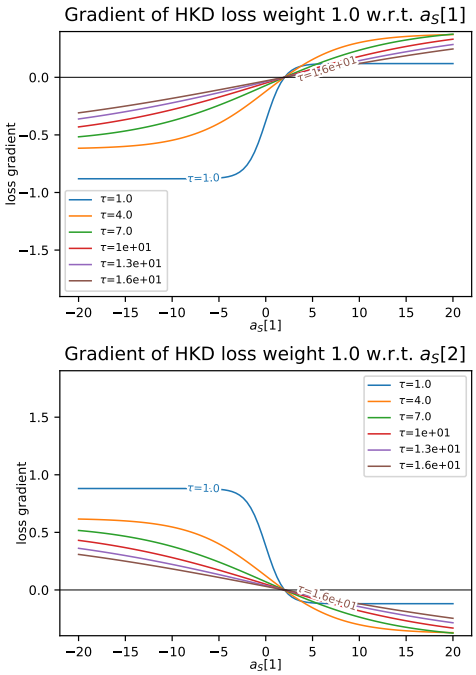



(a) $\tau = 1$, $z_T = (1, -1)$ (b) $\tau = 4$ (c) $\tau = 20$

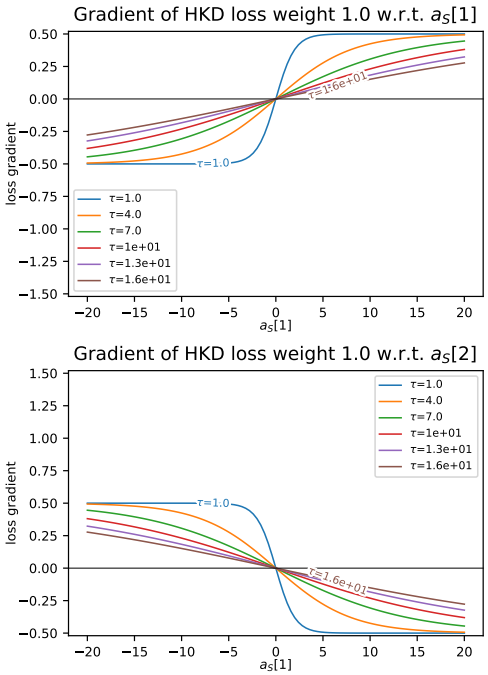



(d) $\tau = 1$, $z_T = (1, 1)$ (e) $\tau = 4$, $z_T = (1, 1)$ (f) $\tau = 20$, $z_T = (1, 1)$

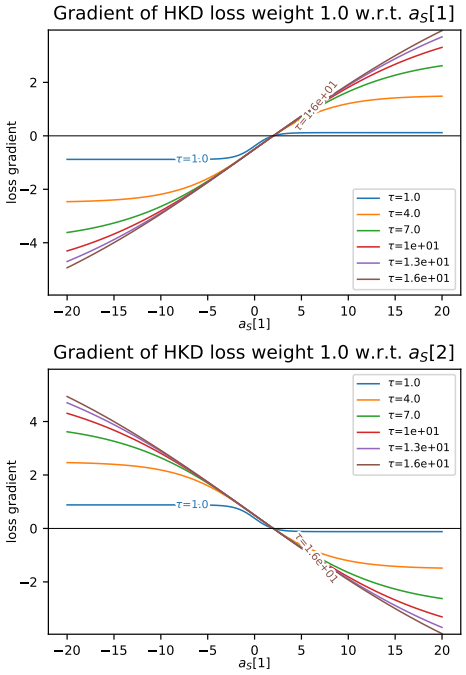
Figure 3.8: Gradients of the HKD loss, w.r.t. the two components of the student logit, for different temperature. The left column has $\tau = 1$, the center column $\tau = 4$, and the right column $\tau = 20$. In the top rows, the teacher has logits $z_T[1] = 1$ and $z_T[2] = -1$, corresponding to $q_T[1] = 0.9$ at $\tau = 1$. In the bottom rows, it has logits $z_T[1] = z_T[2] = 1$, leading to $q_T[j] = 0.5$ for all $\tau$. Note the difference in the horizontal scale, due to the increase in "dynamic range" of the logits when $\tau$ increases. Note that the losses are scaled linearly with the temperature ($\alpha = 1$ in (3.14)), so that the gradients have comparable magnitude.



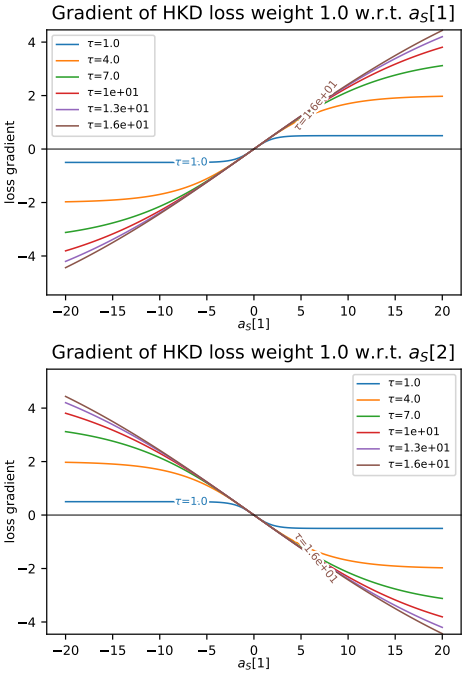

(a) $z_T = (1, -1)$; $q_T^1[1] = 0.9$; $\alpha = 1$




(b) $z_T = (1, 1)$; $q_T^\tau[j] = 0.5$, $j = 1, 2$; $\alpha = 1$




(c) $z_T = (1, -1)$; $q_T^1[1] = 0.9$; $\alpha = 2$




(d) $z_T = (1, 1)$; $q_T^\tau[j] = 0.5$, $j = 1, 2$; $\alpha = 2$

Figure 3.9: Gradients of the HKD loss, w.r.t. the two components of the student logits, for different teachers. Left column: relatively sure teachers ($q_T^1[1] = 0.9$); right: unsure teachers ($q_T = 0.5$ for both components). $z_S[2]$ fixed at 0; the optimum $z_S[1]$ is 2 in the left column, and 0 in the right column. The first row shows the gradients with $\alpha = 1$ in equation (3.15), so that all gradients have the same range. The second row shows the same gradients, but with $\alpha = 2$, demonstrating the equal slope of each curve at the origin.

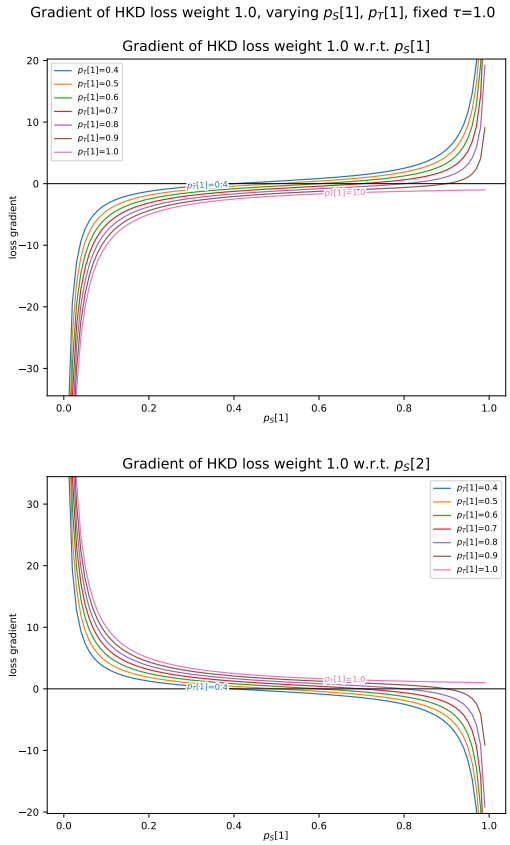



Figure 3.10: Gradient of the HKD, varying $q_S^\tau[1]$ and $q_T^\tau[1]$. Note that these are $q_\cdot^\tau$, not $q_\cdot^1 = q_\cdot$. $\tau = 1$ here, and doesn't influence the curve much. But logits themselves depend on this value.

good prediction) than be positive on the region $z_S[1] \gg z_S[2]$ (corresponding to a too optimistic prediction). This contrasts with what happens if the teacher is unsure with its prediction (equal $p_T$, bottom rows), where the gradient is symmetric relative to the optimum $z_S[1] = z_S[2]$.

- Another effect of the temperature $\tau$ is that this imbalance of intensity of the gradient decreases when $\tau$ increases, even if the teacher is sure with its predictions: the temperature has attenuated the relative difference in class logits, leading to the soften distribution that is usually expected from the knowledge distillation.

- Finally, the gradient has horizontal asymptotes at both extremes for $z_S[1]$: the gradient saturates to a maximum value when the current logit is way off the optimal value.

Note that figure 3.8 uses a linear scaling of the HKD loss ($\alpha = 1$), so that the gradients are comparable.

Figure 3.9 shows the gradient of the HKD loss, when we change $\tau$. When using linear scaling ($\alpha = 1$), we can also see the increase of the size of the linear region with the increase of $\tau$. When using quadratic scaling ($\alpha = 2$), we can see the limiting line that happens when $\tau \to \infty$, with a constant slope.

Figure 3.10 shows the gradient, when we vary $q_S^\tau[1]$ and $q_T^\tau[1]$. The curve is an hybrid of the CE gradient curve: when $q_S^\tau[1]$ is close to one of the boundaries, the curve approaches $\frac{1}{q_S^\tau[1]}$ or $\frac{1}{q_S^\tau[1]-1}$. We can also see that as the teacher becomes confident, the gradient move upward and becomes more and more positive.

### 3.4.3 Combining both losses

When combining the two losses, a choice has to be made regarding the relative weight of the two losses in the total, i.e. one needs to choose a value for $\lambda_{HKD}$. For simplicity, we set $\lambda_{HKD} = 1$ in the following analysis, and use $\alpha = 1$.

Figure 3.11 shows the gradient components, with a relatively sure teacher. For $\tau = 1$, the HKD gradient and the CE gradient are almost equal, up to a constant, so the full gradient keeps the same profile, but two times stronger. For $\tau = 4$, the increase in dynamic range of the HKD gradient causes the CE gradient to have its dynamic range relatively shrinked. And for $\tau = 20$, the increase is so large that the combined gradient seems to "jump" from $-1$ to 0 around $z_S[1] = z_S[2]$. Seen from the other perspective, when we add the HKD loss and gradient, the gradient has an additional positive component when $z_S[1] \gg z_S[2]$ that increases slowly with $z_S$, and an additional negative component when $z_S[1] \ll z_S[2]$ that increases slowly.

Figure 3.12 shows the gradient components with an unsure teacher. The previous remarks still hold.

## 3.5 Analyzing the output of teachers

In this section as well as in the next chapter, we will analyze distributions of the outputs of the networks, in order to determine the regime of knowledge distillation the network is in. For this, we will use graphs generated from TensorBoard's distributions plugins, which allows to provide a set of values and get a graph of the distributions.

Figure 3.13 is one such graph. It shows the distribution of some variable (here, called softmax/predicted wrong) over the training of the network (once per epoch). There are several lines, each one corresponding to a particular quantile of the distribution. The thick blue line in the center of the darkest blue region corresponds to the median of the distribution. The most outer lines delimit the minimum and maximum values of the distribution. The other lines delimit other quantiles; however, the TensorBoard documentation does not provide information on the values used by these other quantiles. By testing various distributions, we can determine that, for instance, the outer two blue lines that delimit the dark blue region correspond to the 31st and 69th percentile respectively.

In the context of knowledge distillation, two sets of distributions are particularly interesting: the distribution of raw outputs from the model, and the distribution of the outputs from the softmax function applied to these raw outputs.

Both outputs have multiple components (one per class), so we focus on two particular components: the component related to the target class (the true label), and the one related to the predicted class (for which the component is highest). Depending on whether the network predicted the correct class or not, this gives 5 distributions of interest:

1. The distribution of the output for the component of the predicted class, regardless of the correctness of the prediction; referred to as "predicted".

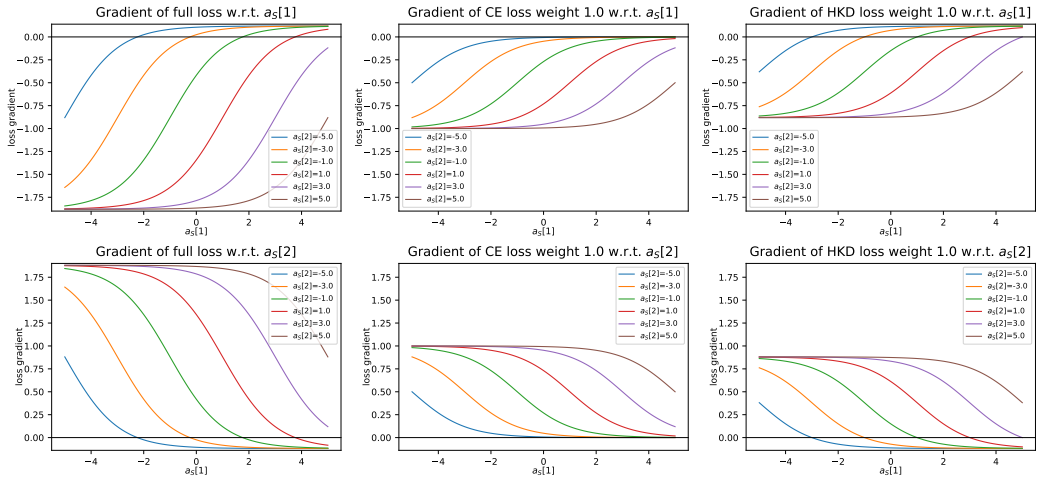
2. The distribution for the predicted class, when the prediction is incorrect; referred to as the "predicted wrong" distribution for the output.

3. The distribution for the target class, regardless of the correctness; referred to as "target".

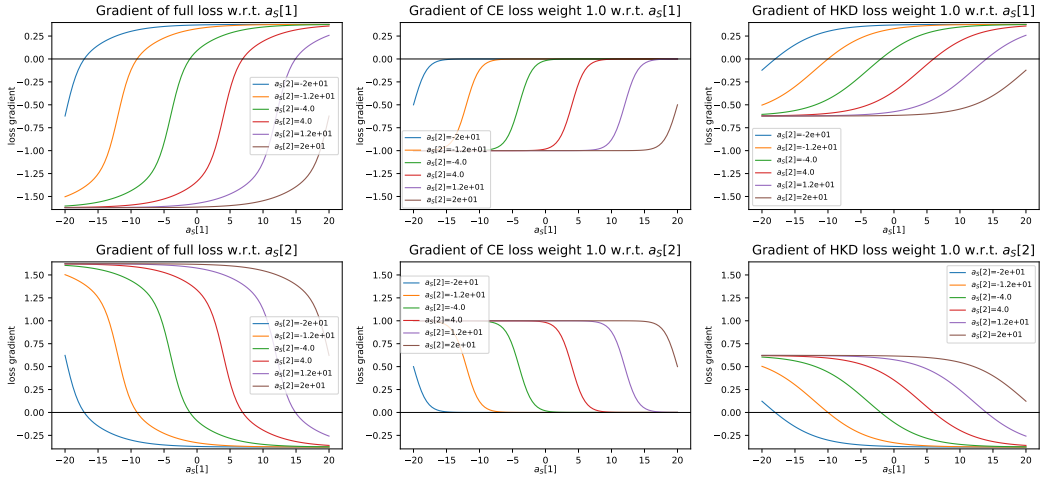
4. The distribution for the target class, when the prediction is incorrect; referred to as "target wrong".

5. The distribution for the predicted class, when the prediction is correct; in this case, this is also the distribution for the target class. We refer to this as the "correct" distribution.

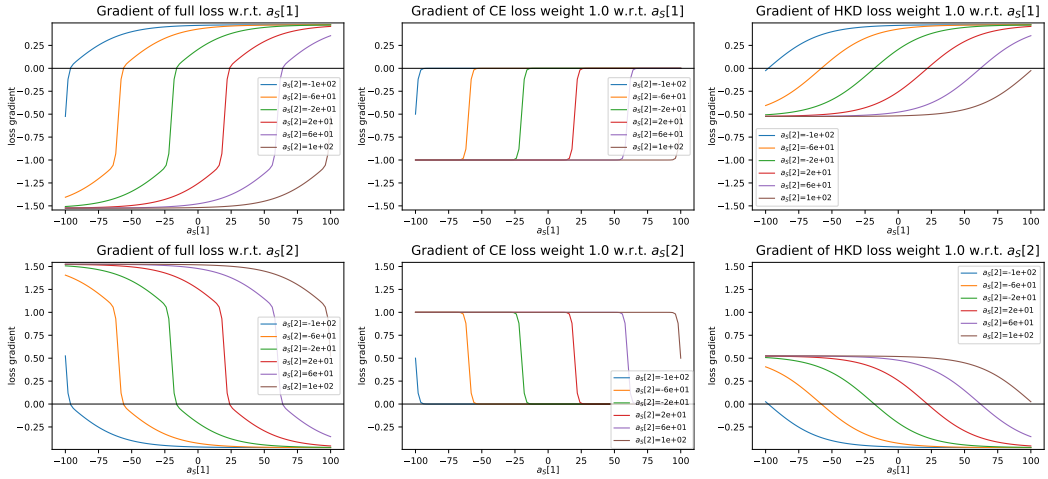
When factoring in the two possible outputs (raw or softmax), there are thus 10 different distributions that can be analyzed. For instance, figure 3.14 show all 10 distributions for a ResNet20-64 network.




(a) $\tau = 1$, $z_T = (1, -1)$

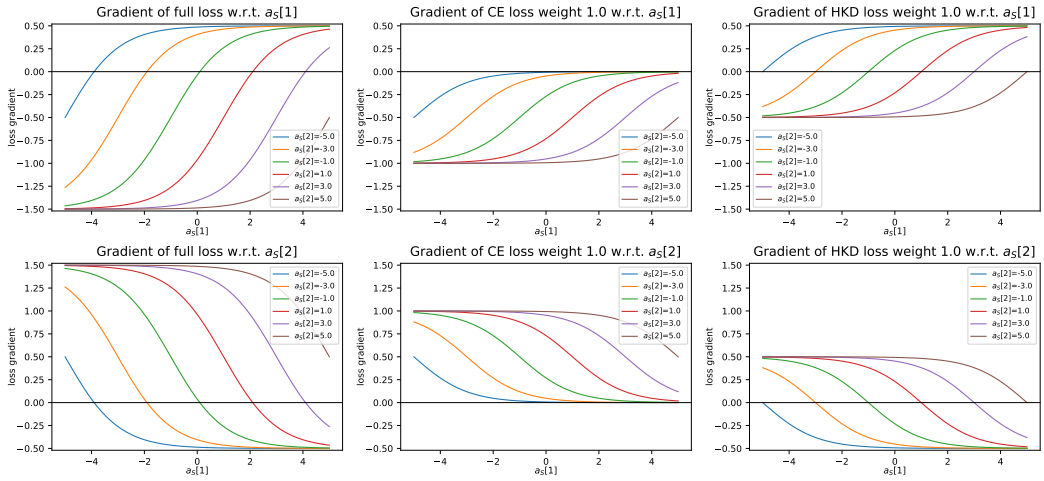



(b) $\tau = 4$

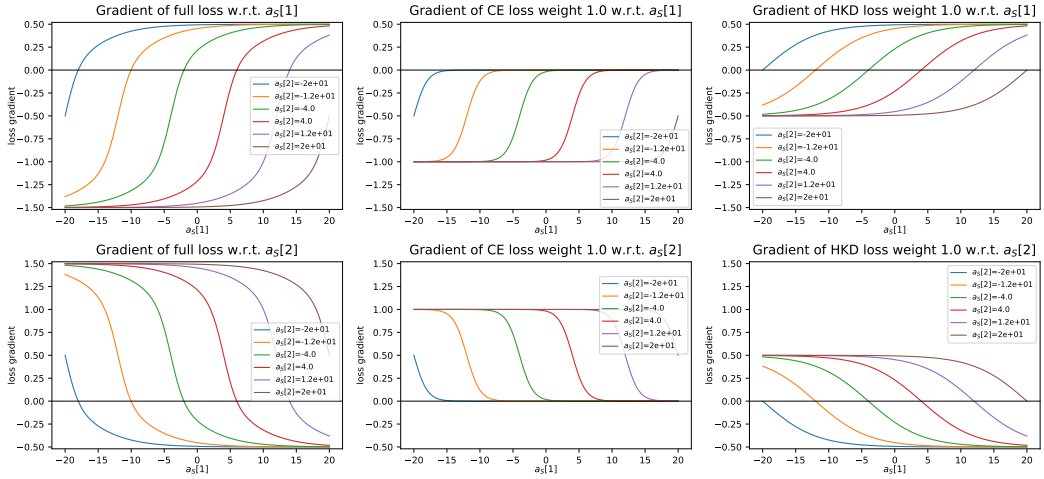



(c) $\tau = 20$

Figure 3.11: Gradients of the full loss and its components, w.r.t. the two components of the student logit, for different temperature. The teacher is a relatively certain one, with $z_T = (1, -1)$ ($q_T^1[1] = 0.9$). Note the difference in the horizontal scale, due to the increase in "dynamic range" of the logits when $\tau$ increases. Note that the losses are scaled linearly with the temperature ($\alpha = 1$ in (3.14)), so that the gradients have comparable magnitude.




(a) $\tau = 1$, $z_T = (1, 1)$




(b) $\tau = 4$, $z_T = (1, 1)$

Figure 3.12: Gradients of the full loss and its components, w.r.t. the two components of the student logit, for different temperature. The teacher is a relatively *unsure* one, with $z_T = (1, 1)$ ($q_T^1[1] = 0.5$).

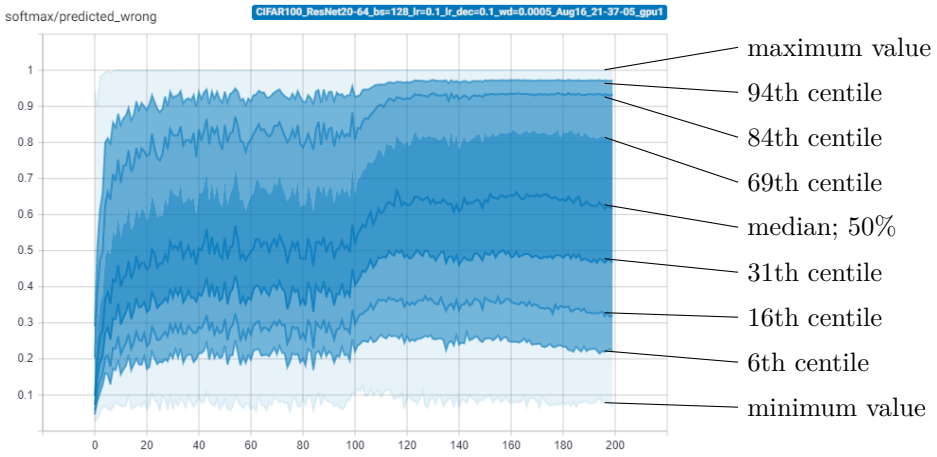
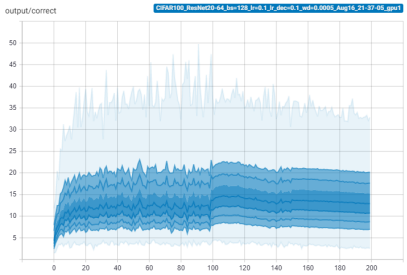



Figure 3.13: Typical graph of the distribution of some variable (here, the output of the softmax for the predicted class when this prediction is wrong) over the training of the network (here, a ResNet20-64). See text for a detailed explanation.

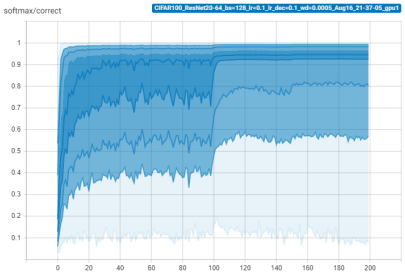
The figure 3.14 thus shows the outputs of one particular network, ResNet20-64. As can been seen, the output values are rather large compared to what the previous section assumed: the values for the predicted class have a range of 2 to 50, with a large number of values in the smaller range 5 – 20. The median value is 12. Predictions when the model is wrong are a bit lower, with a range of 3 – 30 with 90% of values in 5–14, and the median value is 8: this is 4 units lower than the predicted value when the prediction turns out to be correct. When the network makes the wrong prediction, the output for the target label has a range of -2 to 25 (90% of values in 1–10) and a median of 5: this is much lower than the (wrong) predicted value, meaning that the network pretty much gives up trying to provide a meaningful value for these samples.

For the softmax outputs (right column), we can see that the median value for the prediction is about 0.94, with 80% of the values above 0.6, and only 6% of the values below 0.4. When it gets its prediction wrong, the softmax output is still at 50% above 0.6. Interestingly, the values of the softmax for the target class are only at 70% above 0.65; this is relatively normal and expected, as the accuracy of this network is around 77%.

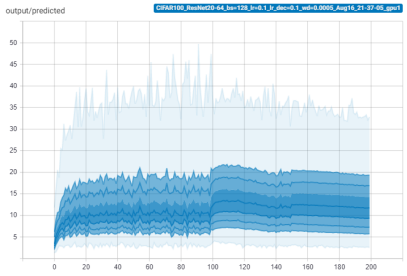
(a) Raw, predicted class, correct prediction



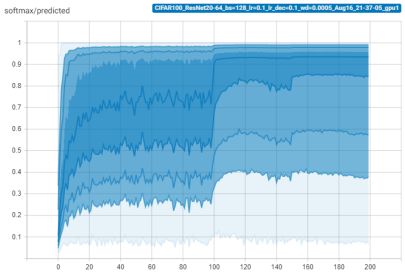

(b) Softmax, predicted class, correct prediction



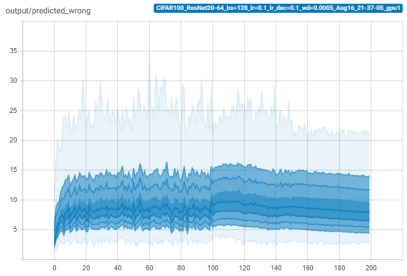

(c) Raw, predicted class



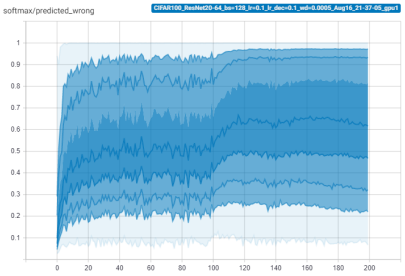

(d) Softmax, predicted class



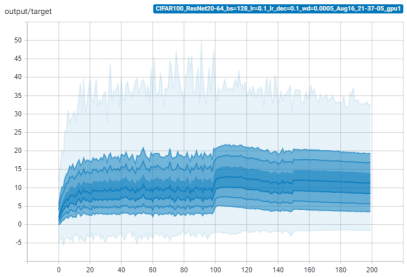

(e) Raw, predicted class, wrong prediction



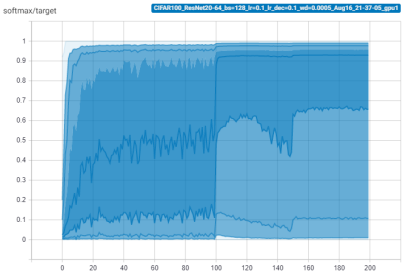

(f) Softmax, predicted class, wrong prediction



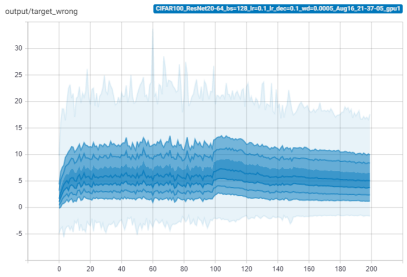

(g) Raw, target class




(h) Softmax, target class




(i) Raw, target class, wrong prediction




(j) Softmax, target class, wrong prediction

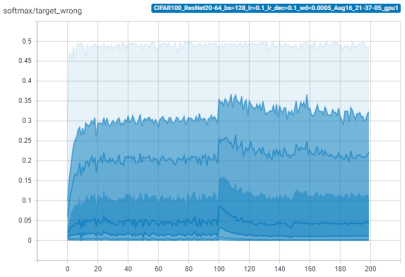



Figure 3.14: All 10 possible output distributions for a particular network (here, a ResNet20-64).

# Chapter 4

# Knowledge distillation applied to FPGA-optimized student architectures

In this chapter, we present experimental results on knowledge distillation, with the goal of determining what parameters affect its effectiveness.

Section 1 presents the experimental framework: the library used to implement the experiments, the training setup and the chosen models.

Section 2 deals with the design of teachers for the subsequent experiments, and how the final students were selected. The student architectures that will be trained are then proposed in section 3.

Finally, section 4 presents the results. First, we determine which distillation hyperparameters are optimal. Then, we conduct experiments to determine the impact of the choice of teacher on the distillation results, and the potential relationships between the student architecture and its distillation accuracy.

## 4.1 Experimental setup

The code for the experiments uses the PyTorch framework [40], version 1.4. It was forked from Yerlan Idelbayev's GitHub repository [26], which provides code to reproduce the experiments from [17] on CIFAR-10 with the ResNet architectures designed for CIFAR-10, which differ from the ResNet architectures used for ImageNet. The Git repository is available at `https://github.com/Jimvy/Thesis-main`.

The experiments were performed on the CIFAR-100 dataset [28], which consists of 50,000 training RGB images of $32 \times 32$ pixels split across 100 classes. This is a more challenging dataset than CIFAR-10, so that differences in accuracies during distillation can be more easily detected, but is less compute-intensive than ImageNet, so that the development and the experiments could be iterated quickly.

The training set of 50,000 images was split into two sets, one of 45,000 images that is used for training the networks, and the other one with the remaining 5,000 images, used as validation set. This validation set is used to select the model parameters that will be saved a the end of a training run, as well as to compare models and choice of hyperparameters. The test set of CIFAR-100, consisting of 10,000 images, was only used at the very end of the experiments, to determine the actual accuracy of the models. It was never used to determine the best hyperparameters or select models for the experiments, and as such its use to evaluate the true accuracy of the models is relatively unbiased.

We use PyTorch's SGD as optimizer, which performs mini-batch stochastic gradient descent with momentum. The momentum was fixed at 0.9 for all experiments, the mini-batch size at 128.

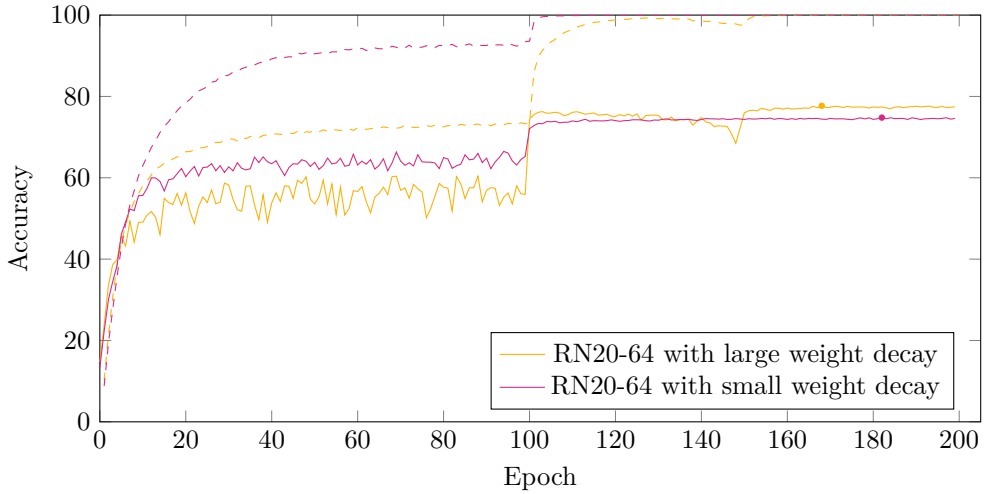



Figure 4.1: Learning curves for ResNet20-64, with either large weight decay $5 \times 10^{-4}$, or small weight decay $1 \times 10^{-4}$. Continuous lines for validation accuracy, dashed lines for training accuracy, the dot marks the best accuracy and thus the accuracy of the saved model.

The initial learning rate was kept as an hyperparameter, chosen among 0.05, 0.1 and 0.2.

All experiments run the training for 200 epochs, and use a time-based learning rate schedule, with division by 10 of the learning rate at epochs 100 and 150; that is, it performs 100 epochs of training at the specified initial learning rate $\eta$, then 50 epochs at $\eta/10$ and a final 50 epochs at $\eta/100$. This is close to the original schedule of [17], as $\frac{45{,}000}{128} \cdot 50 \approx 17,500 \approx 16,000$. Another schedule was tried, with decay steps at epochs 60, 120 and 160, but it didn't provide significant improvements over the original learning rate schedule.

During preliminary experiments, we investigated warming up the learning rate, as suggested in He et al [20] and Goyal et al [13], by implementing the simple learning rate warmup implemented in [17]. This usually lead to networks with higher accuracy (around 1-2%), especially for deeper networks like ResNet44 and ResNet56. However, the final experiments don't use learning rate warmup, as the goal is not to have the highest-performing networks, but to evaluate the efficacy of knowledge distillation compared to its absence.

For regularization, we used the $l_2$ norm or *weight decay*. We initially tested a standard weight decay of $1 \times 10^{-4}$, but also tested a weight decay of $5 \times 10^{-4}$, which tends to drastically improve the performance of the networks by 3-5%. However, it often lead to a strange learning curve, as in figure 4.1: usually, we notice a bump in accuracy at epochs 100-120, but then a continuous decline in accuracy at epochs 120-150, followed again by a bump in accuracy. Some networks have their highest accuracy after epoch 150 during this second bump, but others did not reach the same accuracy in this second bump and had their best accuracy before epoch 120. This is a very similar effect to what happens when the learning rate is too high, which is not surprising as increasing the weight decay actually increases the effective learning rate, when the network has batch normalization (see Hoffer et al [23]). It also turned out to be a red herring when applied to knowledge distillation. In the following sections, the $1 \times 10^{-4}$ weight decay is referred to as "small" or "low weight decay", and $5 \times 10^{-4}$ is referred to as "large" or "high weight decay".

The networks used are all Residual Networks (ResNets), from [17], with the architectures designed for the CIFAR-10 dataset. However, such networks perform poorly on the CIFAR-100 dataset due to the limited number of channels per layer, so for most experiments we scale the number of channels for all layers by increasing the *base width* (the number of output channels of the first layer and residual blocks, when the output feature map is $32 \times 32$) from the default 16 channels to the new number of channels; all other layers have their number of channels multiplied by the same factor. In the next sections, we will abbreviate "ResNet" with "RN". All model parameters are default float 32 numbers; we don't consider quantization or fixed-point numbers

| Name | # weight layers | #params per base width | | |
|---|---|---|---|---|
| | | 16 | 32 | 64 |
| ResNet20 | $20 = 1 + (3 + 3 + 3) \cdot 2 + 1$ | 275k | 1.08M | 4.31M |
| ResNet32 | $32 = 1 + (5 + 5 + 5) \cdot 2 + 1$ | 470k | 1.86M | 7.41M |
| ResNet44 | $44 = 1 + (7 + 7 + 7) \cdot 2 + 1$ | 664k | 2.64M | 10.6M |
| ResNet56 | $56 = 1 + (9 + 9 + 9) \cdot 2 + 1$ | 859k | 3.41M | 13.7M |
| ResNet34 | $34 = 1 + (3 + 4 + 6 + 3) \cdot 2 + 1$ | 1.34M | 5.31M | 21.2M |
| ResNet50[1] | $50 = 1 + (3 + 4 + 6 + 3) \cdot 3 + 1$ | 1.53M | 5.99M | 23.7M |

Table 4.1: ResNet architectures used in the following experiments

in this thesis, leaving this topic as an orthogonal choice and for future research.

To initialize the parameters in the network, we apply Kaiming initialization, to the weights in the convolutional and linear layers, as proposed in [18].

Table 4.1 gives the various architectures used in the experiments. ResNet20, ResNet32, ResNet44 and ResNet56 are all architectures designed for CIFAR. ResNet34 and ResNet50 are adaptations of networks designed for ImageNet, for the CIFAR dataset. Additional details can be found in appendix A.

The main loss function, used for all experiments, is the cross-entropy loss, as defined in the PyTorch library. When using knowledge distillation, we use the expressions from chapter 3, and weight the HKD loss by an hyperparameter $\lambda_{HKD}$ before adding it to the full criterion.

The validation set is used to determine the model that will be saved at the end of a training run: the model that performed best on the validation set (highest valid. accuracy) is saved on disk. The saved model may thus have been saved at any step during its training, and there are quite a few occasions where the best performing model was saved before epoch 120. This is especially the case when the learning rate was too high or when large weight decay values were used.

In some cases, models that performed best up to some epochs were saved on disk before the end of the training, to obtain "early stopped" teachers. This is explained in a later section.

For visualization purposes, we use TensorBoard[2], a component of the TensorFlow framework [1]. TensorBoard allows visualizing learning curves over time, distributions of parameters, precision-recall curves, and analyze hyperparameter choices in a practical graphical interface. Most graphs included in this thesis were generated using it.

### 4.1.1 Issues encountered during the development

During the development of the code for experiments, several issues and possible gotchas were identified.

**Implementing the cross-entropy in knowledge distillation** The cross-entropy loss is used in knowledge distillation to estimate the difference between the teacher output and the student output. However, the cross-entropy loss provided by PyTorch as `CrossEntropyLoss` is designed for the specific case when there is only one target label for the output. There are two different ways of implementing a cross-entropy loss:

- We can use the Kullback-Leibler divergence from the class `KLDivLoss`; this will introduce a bias, but as this bias is fixed by the dataset distribution, it can be ignored.
- We can implement directly the expression of the cross-entropy; this has the advantage of not introducing a bias, but can be slightly slower compared to a native implementation of a `KLDivLoss`.

[1] This network uses bottleneck residual blocks with 3 stacked layers instead of the basic block of 2 stacked layers used by the other networks

[2] `https://www.tensorflow.org/tensorboard`

The second solution was used, as the slowdown is negligible compared to the rest of the computations.

**Set the teacher model into evaluation mode** When loading a second model to serve as a teacher for knowledge distillation, it is important to set that model into evaluation mode (using `teacher.eval()`) to avoid the model from being modified during training. This is especially important if the model has batch normalization layers: in this case, by default the variables holding the mean and variance statistics continue to be updated as new batches are processed by the network, and this may cause variations in the teacher accuracy.

## 4.2 Choice of teachers

CIFAR-100 is a more challenging dataset, but is not the focus of much research, and does not have as many good networks and architectures as with CIFAR-10. So, the first step was to build networks that could perform reasonably well on this dataset and could be used as teachers during distillation for subsequent experiments.

For this, we took the ResNet network architectures designed for CIFAR-10 and increased the number of channels for each layer. The base models have 16 channels in the first convolutional layer and in all layers of the first few residual blocks; we refer to the number of channels in those layers as the *base width*. To produce wider networks, we simply scaled the number of channels in each layer by the same factor $S$. That way, the base width of the networks becomes $\text{BW} = 16 \cdot S$, and we refer to the resulting architecture as a ResNetN-BW, where N is the number of layers in the network. For instance, ResNet32-64 is a ResNet32 where the number of channels in each layer has been multiplied by $64/16 = 4$, compared to the original architecture. This is essentially the technique used in Wide Residual Networks [63], except that we also scaled the initial convolutional layer and did not apply the modifications from [19].

The networks were then trained multiple times to eventually obtain a good model for each architecture. Various initial learning rates were tested, among 0.1, 0.05, 0.2, as well as the two different values for the weight decay. For each architecture, the model with the best validation accuracy was selected as a candidate teacher for this architecture and configuration. We also included networks designed for ImageNet, ResNet34 and ResNet50, in case they turned out to be as effective on CIFAR-100 as they are on ImageNet. The best teachers obtained are given in table 4.2.

To select the teachers used for the subsequent experiments, we plotted their validation accuracy with respect to their number of parameters, as a measure of their capacity and complexity (figure 4.2).

The networks designed for ImageNet, ResNet34 and ResNet50, performed exceptionally bad with regards to their number of parameters. This may be due to a suboptimal training setup, or maybe they are inadequate for this dataset.

ResNet56 tended to perform worse than less deep ResNets with the same amount of channels. Again, this is probably due to the training setup, which uses a relatively "primitive" learning rate schedule. Notably, for the final experiments no learning rate warmup was used, although preliminary experiments had showed it could improve the performance of such a deep network.

Networks in the upper left corner are Pareto optimal, as they have higher accuracy at a lower number of parameters than networks located below or right of them. They are thus the primary choice for teacher networks. We thus select ResNet44-64 (as the best performer overall at 78.4%), ResNet20-64 (Pareto optimal, 77.7%) and ResNet44-32 (thinner teacher than ResNet44-64, 75.4%), among the networks trained with a large value for the weight decay.

For comparison, we also select some teachers among the networks trained with a low but standard value for the weight decay. Indeed, although these networks have a lower accuracy, they have a different distribution of their output values, which may be more useful for the students during knowledge distillation. Also, they tend to not have this strange bump of accuracy around epochs 100-120 followed by a decline for epochs 120-150.

| Architecture | # params | WD | Valid acc (%) | Test acc (%) |
|---|---|---|---|---|
| RN20-64 | 4.31M | Low | 74.86 | 74.55 |
| | | High | 77.68 | 77.26 |
| RN32-32 | 1.86M | Low | 72.46 | 72.13 |
| | | High | 74.34 | 74.16 |
| RN32-64 | 7.41M | Low | 74.72 | 74.37 |
| | | High | 77.8 | 76.87 |
| RN44-32 | 2.64M | Low | 71.96 | 72.52 |
| | | High | 75.36 | 75.11 |
| RN44-64 | 10.5M | Low | 75.08 | 74.83 |
| | | High | 78.36 | 77.94 |
| RN56-32 | 3.41M | Low | 73.38 | 72.65 |
| | | High | 74.52 | 74.32 |
| RN56-64 | 13.6M | Low | 72.62 | 72.68 |
| | | High | 77.36 | 77.69 |
| RN34-32 | 5.31M | Low | 72.66 | 72.69 |
| | | High | 75.68 | 75.50 |
| RN34-64 | 21.2M | Low | 74.1 | 73.76 |
| | | High | 76.94 | 76.99 |
| RN50-32 | 6.00M | Low | 71.76 | 71.38 |
| | | High | 76.94 | 75.83 |
| RN50-64 | 23.7M | Low | 74.4 | 73.92 |
| | | High | 77.92 | 76.99 |

Table 4.2: Best accuracies for networks trained as teachers. The model with highest validation accuracy was selected as the best teacher for its architecture, over a number of runs with different initializations and initial learning rates. Two values for the weight decay (WD) were tested: one with $1 \times 10^{-4}$ referred to as "low", another with $5 \times 10^{-4}$ referred to as "high".

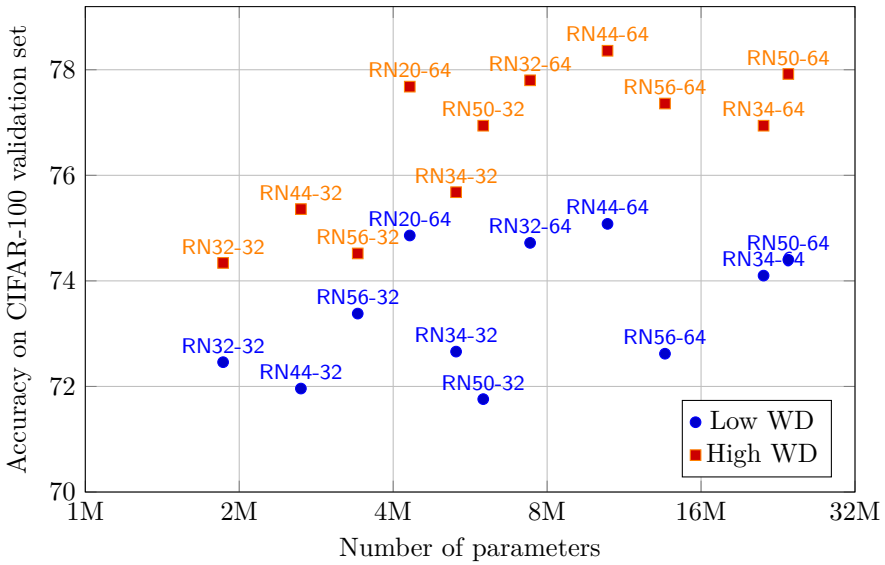



Figure 4.2: Accuracy of the teachers with respect to their number of parameters. Two families of teachers are presented: one trained with a low weight decay of $1 \times 10^{-4}$, the other one with a large weight decay of $5 \times 10^{-4}$.

Inspired by the results of Cho et al [7], we also include as potential teachers some models which were trained up to epoch 120 only, instead of for the full 200 epochs. Indeed, according

| Arch. | # params | Weight decay | LR | Valid acc (%) | Test acc (%) |
|---|---|---|---|---|---|
| ResNet44-64 | 10.5M | $5 \times 10^{-4}$ | 0.1 | 78.36 | 77.94 |
| ResNet20-64 | 4.3M | $5 \times 10^{-4}$ | 0.1 | 77.68 | 77.26 |
| ResNet44-32 | 2.6M | $5 \times 10^{-4}$ | 0.05 | 75.36 | 75.11 |
| ResNet44-64 | 10.5M | $1 \times 10^{-4}$ | 0.05 | 75.08 | 74.83 |
| ResNet32-32 | 1.86M | $1 \times 10^{-4}$ | 0.1 | 72.46 | 72.13 |
| ResNet20-64 | 4.3M | $5 \times 10^{-4}$ | 0.1 | 76.26 | 76.11 |
| ResNet44-64 | 10.5M | $5 \times 10^{-4}$ | 0.1 | 74.26 | 73.96 |

Table 4.3: Selected networks for use as teachers in the following experiments. For the last three models, their training was "stopped" at epoch 120, and the model with best validation accuracy up to this epoch was saved to disk.

| Architecture | #params | Valid acc (%) | Test acc (%) |
|---|---|---|---|
| ResNet20-16 | 276k | 66.6 | 66.1 |
| ResNet20-24 | 614k | 69.6 | 68.8 |
| ResNet20-32 | 1.1M | 70.8 | 70.9 |
| ResNet32-16 | 470k | 67.7 | 68.1 |
| ResNet32-24 | 1.1M | 70.6 | 70.6 |
| ResNet44-16 | 664k | 68.5 | 68.3 |
| ResNet44-24 | 1.5M | 69.2 | 69.4 |
| ResNet56-8 | 218k | 62.9 | 62.7 |
| ResNet56-16 | 859k | 68.9 | 67.7 |

Table 4.4: Student architectures used in the experiments. Accuracy shown is the median accuracy of three runs each trained with initial learning rate 0.1 and small weight decay.

to them, large networks, whose training is stopped early, tend to behave like networks with smaller capacity, suffer from less overfitting, and have showed in their experiments to consistently perform better as teachers, i.e. to train students with higher accuracy. The early stopping was implemented in a rather rough way: during the full training of 200 epochs of a model, the model with the highest validation accuracy between the epochs 100 to 120 was saved to disk, in addition to the overall best model. Epoch 100 was chosen because it was just after the learning rate had been dropped by a factor of 10, and just after a large increase in the validation accuracy of the model.

Overall, the selected teachers are given in table 4.3, along with their test set accuracy (which was never used for the selection decision).

## 4.3 Choice of students

The choice of student architectures follows the same principles. We scaled the number of channels for each layer to have networks of different capacity and complexity. Table 4.4 gives the different student architectures that are tested, along with their validation and test accuracy. Those networks were chosen to be much smaller than the considered teachers, and to be likely more amenable to an efficient implementation on an FPGA.

## 4.4 Results

In this section, we present the results of our experiments to evaluate and analyze the practical use of knowledge distillation.

We want to evaluate the possibility of using knowledge distillation to train small networks

adapted to embedded execution. For this, we will evaluate the three components of knowledge distillation:

1. The hyperparameters used during knowledge distillation: common choices are $\tau = 4$ and $\tau = 6$, with a myriad different values for the weight (as the notations differ from one paper to the next). We would like to identify which set of distillation parameters works best in general.

2. The effectiveness of knowledge distillation on students: to what extend can knowledge distillation close the gap between the accuracy of a student trained alone and the accuracy of a good teacher? And does it depend on the capacity of the student?

3. The teachers used: as in real life, not all teachers are equally good at giving a course. It would be interesting to see if there are common properties that good teachers should have.

### 4.4.1 Optimal parameters for knowledge distillation

The first experiment aims to determine a good range of values for the hyperparameters used in knowledge distillation: the temperature $\tau$ and weight $\lambda_{HKD}$.

As it is infeasible to test every possible combination of hyperparameters (temperature, weight, initial learning rate) and students/teachers, for this experiment we settled on a single pair of teacher/student. The teacher is the 75.3%-accurate ResNet44-32, trained with a large weight decay of $5 \times 10^{-4}$ and with 2.6M parameters. The students are ResNet32-16, trained with an initial learning rate of 0.1 and a standard weight decay of $1 \times 10^{-4}$, with 470k parameters. Trained without distillation, this network achieves a median validation accuracy of 68.4% (and a test accuracy of 68.1%), and is the baseline for this section.

We tested various choices of temperature and weights. Each configuration was run twice and the median of the accuracies of each run[3] is taken as the accuracy for this configuration. The results are presented in table 4.5.

| | Temperature | | | | | |
|---|---|---|---|---|---|---|
| Weight | 1 | 2 | 4 | 6 | 8 | 16 |
| 0.02 | 68.3 | 68.3 | 68.5 | 68.9 | 69.1 | 68.6 |
| 0.1 | 68.3 | 68.2 | 70 | 70.2 | 69.3 | 69.4 |
| 0.2 | 68.4 | 68.8 | 70.5 | 70.1 | 70.2 | 69.8 |
| 0.5 | 67.1 | 69 | 71.4 | 71.2 | 70.6 | 70.3 |
| 1 | 67.1 | 70.1 | 70.6 | 71.2 | 71 | 71.3 |
| 2 | 67.2 | 69.5 | 71.2 | 71.6 | 71.2 | 71.2 |
| 5 | 64 | 68.2 | 70.7 | 70.7 | 71.7 | 71.8 |
| 8 | 63.6 | 34 | 68.8 | 69.5 | 71.2 | 70.9 |

Table 4.5: Knowledge distillation from ResNet44-32 into ResNet32-16, with different values for the temperature and weight. The median accuracy on the validation set of two runs are presented.

From these results, we can draw a few conclusions. First, distilling with a small weight (0.02) only gives a slight improvement (less than 0.5%) over the baseline. This is essentially the same as training without knowledge distillation.

Strangely, distilling with $\tau = 1$ actually *decreases* the accuracy of the student over the baseline, and this effect increases as the weight increases.

Larger temperatures and larger weights lead to higher accuracies, which can reach a full 3% above the baseline, or the same accuracy of a network with double the number of channels.

No combination of temperature and weight managed to provide the same accuracy as the teacher, as they seem limited to about 71.8%. Interestingly, this is roughly the same accuracy as a ResNet44-32 teacher trained with a weight decay of $1 \times 10^{-4}$.

[3]By accident, some configurations ($\tau = 6$ and $\lambda_{HKD} = 0.02$ or 0.1) were run more than twice.

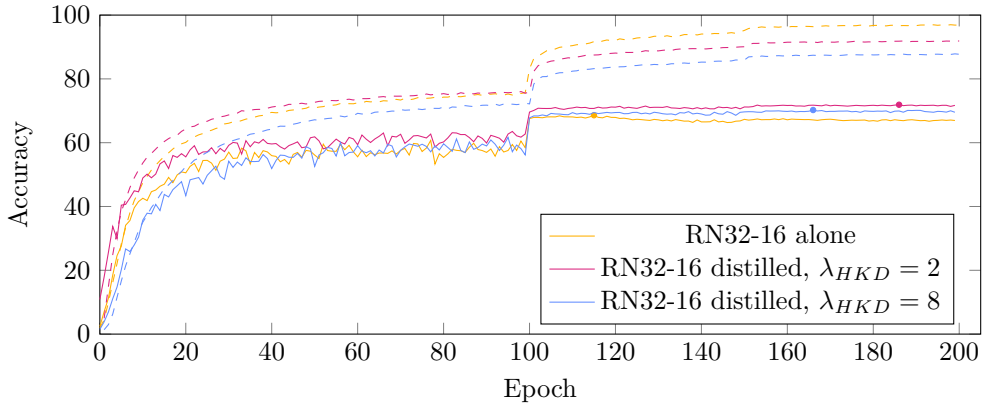



Figure 4.3: Typical learning curves for 3 configurations: ResNet32-16 student trained without distillation; student trained with the ResNet44-32 teacher at temperature 6 and weight 2; and student trained with weight 8. Continuous lines for validation accuracy, dashed lines for training accuracy, the dot marks the best accuracy and thus the accuracy of the saved model.

We also note that very large weights can lead to issues during training. For instance, the 34% median accuracy for the $\tau = 2$, $\lambda_{HKD} = 8$ configuration is due to one run that failed to converge. Also, the convergence during training is slightly slowed down initially, as shown on figure 4.3. The training accuracy when using $\lambda_{HKD} = 8$ (blue curve) initially stays at a few % for the first few epochs, in contrast with a training with $\lambda_{HKD} = 2$ or without distillation where the training accuracy is much quicker to start converging.

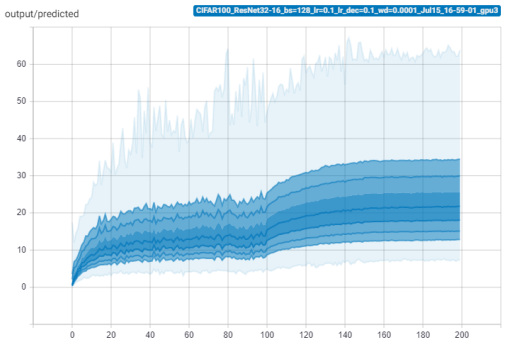
An interesting behavior appears when looking at the output distributions of the networks. Figure 4.4 show some of the distributions for two networks, a student trained without distillation, and a student trained with distillation.

The first row shows the distribution for the raw outputs, for the class predicted by the model. The range of values is quite different: the student trained alone has a final distribution with a median value of around 22 and a range of 8 – 65, while for the student trained with distillation the median is around 13 with a range of 3 – 30. Over the course of the training, the distribution for the student trained alone constantly shift towards larger values, while the distilled student has a distribution that mostly stays in the same range. The distilled student distribution is also very close to the distribution from the teacher. Knowledge distillation seems to have a normalization effect on the output distributions, even though the output range is never directly affected by the distillation loss.

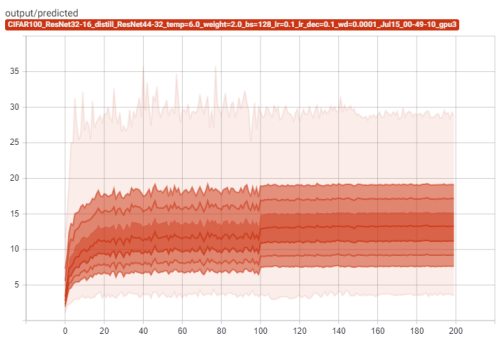
Another observation is that, for the distilled student, the distributions vary much less across training. During the epochs 20 to 99, the raw output distribution doesn't change much, and during epochs 100 to 200 the distribution is roughly constant. In comparison, the non-distilled student sees its distributions constantly shifting to larger values, except during the last 50 epochs. This phenomenon is also visible on the graphs of the distributions of the softmax output for the predicted class, when this prediction turns out to be wrong: the distribution for the distilled student is relatively stable from epoch 100 onward and between epochs 40 to 100, whereas for the non-distilled student the distribution shifts for most of the training. This is quite surprising, given that the validation accuracy of both networks doesn't change much from epoch 100 onward, as show in figure 4.3: while the accuracy stays the same, the output of the non-distilled model continues changing and shifting, while the distilled model has a more stable distribution of its output.

These relatively stable distributions during the majority of the training, along with the relatively stable learning curves, suggest that we could possibly reduce the training time when applying knowledge distillation: it is possible that knowledge distillation speeds up the initial training, but then starts to hold back the student. This may be an interesting future experiment.

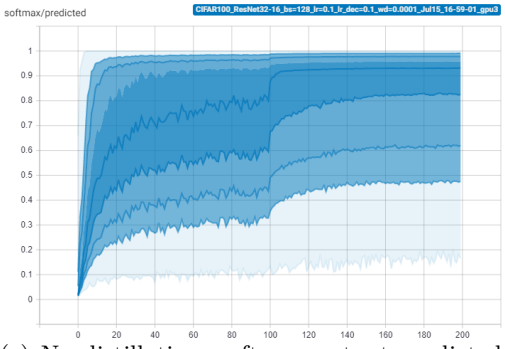
(a) No distillation; output predicted



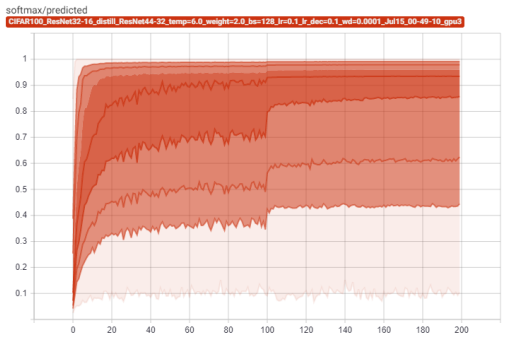

(b) With distillation; output predicted



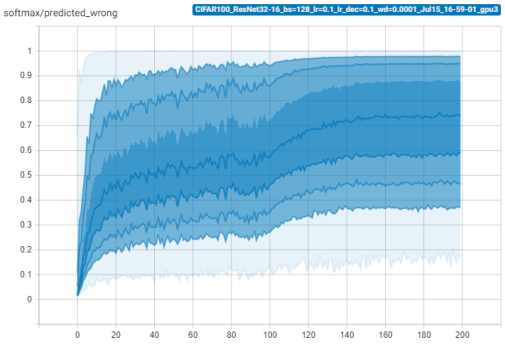

(c) No distillation; softmax output predicted




(d) With distillation; softmax output predicted




(e) No distillation; softmax output predicted when the prediction is wrong




(f) With distillation; softmax output predicted when the prediction is wrong

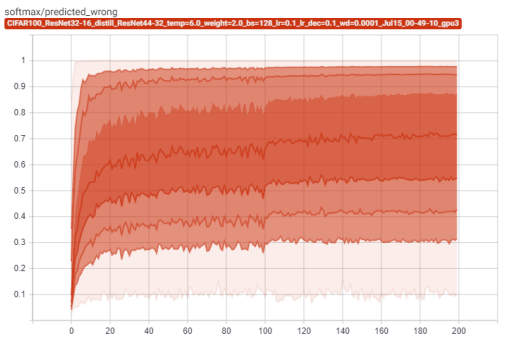



Figure 4.4: Distributions of the student model outputs. Left column: student trained without distilation. Right column: student trained with ResNet44-32 as a teacher, with $\tau = 6$ and $\lambda_{HKD} = 2$. See subsection 3.5 for an explanation of what this means.

### 4.4.2 Which students can be best trained using knowledge distillation?

For the following experiments, a temperature $\tau = 6$ and a weight $\lambda_{HKD} = 2$ will be used. Although not the best configuration with regards to the results of the previous section, it is close to the choice of parameters that is usually chosen in the literature.

Now that we have a good set of values for the distillation hyperparameters, we can turn our attention on the second question: how well can smaller students be trained with knowledge distillation?

Figure 4.5 shows the results, for four different teachers.

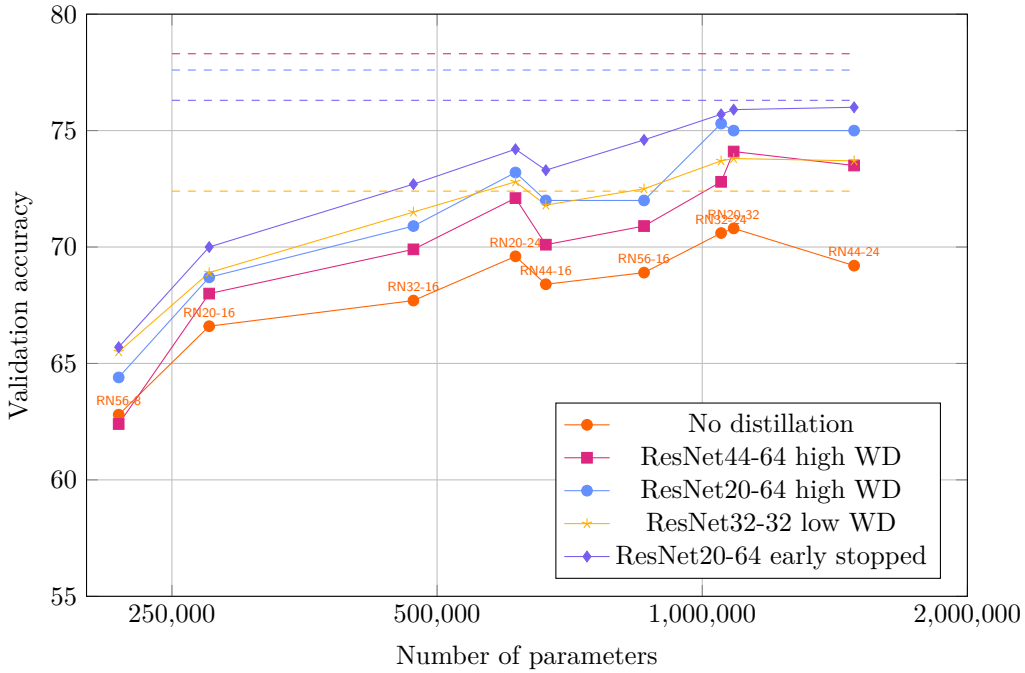



Figure 4.5: Distillation with students of various capacity, and with different teachers. All students are trained with initial learning rate 0.1 and low weight decay. The dashed lines represent the accuracy of each teacher.

The overall trend is that with increasing capacity, the accuracy of the students increases, which should come as no surprise. However, the accuracy of the ResNet44-16, ResNet44-24 and ResNet56-16 is markedly lower than expected, which may be due to a suboptimal training of these deep networks.

In terms of distillation, all teachers tend to improve the accuracy of their students, but there are large differences between the teachers. Surprisingly, students seem to have no particular preference for the teacher that helps them during training: Aside from the relative performance of the teacher, and from the base performance of the student, there is not much impact on the accuracy from a potential match or mismatch of some characteristics of the architectures used for the teacher or the student.

Additional results are available in figure B.1, in the appendix.

### 4.4.3 Which teachers are most effective for a student?

The last important point of knowledge distillation is the choice of the teacher.

Table 4.6 gives the accuracy of a sample of students, trained with a variety of teachers, split into three categories: large value for the weight decay during training, small value for the weight decay, and early stopped (less than 120 epochs). Along with figure 4.5, this allows to estimate the relative performance of teachers.

| | | high WD | | | low WD | | early stopped | |
|---|---|---|---|---|---|---|---|---|
| Student | Alone | 44-64 | 20-64 | 44-32 | 44-64 | 32-32 | 20-64 | 32-32 |
| RN20-16 | 66.6 | 68 | 68.7 | 68.9 | 68.5 | 68.9 | 70 | 70.1 |
| RN20-24 | 69.6 | 72.1 | 73.2 | 73.2 | 72.8 | 72.8 | 74.2 | 73.8 |
| RN20-32 | 70.8 | 74.1 | 75 | 74.7 | 74 | 73.8 | 75.9 | 75.6 |
| RN32-16 | 67.7 | 69.9 | 70.9 | 71.4 | 70.3 | 71.5 | 72.7 | 72.3 |
| RN32-24 | 70.6 | 72.8 | 75.3 | 74.3 | 73.6 | 73.7 | 75.7 | 74.9 |
| RN44-16 | 68.4 | 70.1 | 72 | 72.4 | 71.5 | 71.8 | 73.3 | 73.6 |
| RN44-24 | 69.2 | 73.5 | 75 | 74.8 | 73.9 | 73.7 | 76 | 75.9 |
| RN56-8 | 62.8 | 62.4 | 64.4 | 65.2 | 64.4 | 65.5 | 65.7 | 65.8 |
| RN56-16 | 68.9 | 70.9 | 72 | 73.1 | 72.5 | 72.5 | 74.6 | 73.6 |
| Teachers | NaN | 78.3 | 77.6 | 75.4 | 75.1 | 72.4 | 76.3 | 74.3 |

Table 4.6: Knowledge distillation with varying teachers. Column names give the depth and the base width of the network. For each of the runs, $\tau = 6$, $\lambda_{HKD} = 2$ and $\text{lr}_{init} = 0.1$. Shown is the median validation accuracy of two runs.

| Student | Alone | Trained with | | |
|---|---|---|---|---|
| | | ResNet32-32 | ResNet20-64 | ResNet44-64 |
| ResNet20-16 | 27.6 | 40.4 | 45.9 | 73.7 |
| ResNet20-32 | 37.3 | 53.2 | 60.0 | 88.1 |
| ResNet32-16 | 38.6 | 52.7 | 57.5 | 84.1 |
| ResNet44-24 | 62.2 | 78.9 | 85.6 | 111 |

Table 4.7: Training times of students, with or without distillation, with various teachers. Time in minutes.

First, we can see that both ResNet44-64 are relatively bad teachers: students trained with smaller teachers are almost always more accurate than when trained with a ResNet44-64, even though the ResNet44-64 have higher accuracy. This may be due to inappropriately trained teachers, or maybe due to the large difference in capacity between the teacher and its students.

Next, the ResNet44-64 teacher, trained with a small weight decay, outperforms the ResNet44-64 teacher trained with a large weight decay. ResNet32-32, trained with a small weight decay, is also competitive against the ResNet44-32 teacher, while having 35% less parameters. It can also train students that actually surpass their teacher, a behavior already observed in the literature.

The best teachers are however the networks whose training was stopped early: their students often reach a full 1% higher accuracy than students trained with any other teacher, and sometimes even 2%. Even more, some students have surpassed their teacher in terms of accuracy, such as the ResNet20-32 trained with ResNet20-64, or the ResNet44-24 trained with this teacher too. This coincides with the observations from [7].

Finally, a note on the training time required for distillation. Table 4.7 gives the training time required to train some students, either alone or with knowledge distillation. The smallest teacher architecture tried, ResNet32-32, has a relatively small impact on the training time, around 25%-40%. However, distilling with ResNet20-64 requires as much as 35%-60% more time, and distilling with ResNet44-64 requires around 2 times more time to train, and proved to have mixed results.

In our context, having a longer training time when using knowledge distillation is not too problematic, as the goal is to obtain a small, efficient network for embedded applications that is as accurate as possible. However, if the deployed model needs to be frequently updated (for instance, with a better model trained on a better dataset), then this longer training time can become problematic during development of the model.

# Chapter 5

# Conclusion

Deep learning has seen a huge increase in interest and applications in the last decade. But, in order to achieve its spectacular performance, it also requires vast amounts of computational power, storage requirements, and its models are difficult to deploy on embedded platforms, like FPGAs. Many techniques have been proposed with the goal of reducing the computational requirements of deep learning models. Among them, knowledge distillation proposes to train a small network, called a student, by giving it access to a larger and more accurate teacher model, and training the student to reproduce, to some extent, the outputs of the teacher. This method has proved to lead to higher accuracy students, that can even surpass their teachers in some circumstances.

In this thesis, we have tried to evaluate how knowledge distillation can be used in practical settings, with the goal of training small networks adapted for FPGAs.

Therefore, we have conducted experiments to compare the accuracy of students trained with or without distillation. We used ResNets and Wide ResNets on a classification task with the CIFAR-100 dataset, with a basic but sufficient training setup. We have selected a few teachers based on their accuracy and capacity.

A first experiment was conducted to determine a good set of values for the temperature and weight hyperparameters. Our experiments gave us $\tau = 6$ and $\lambda_{HKD} = 2$ as a good starting point, and we noted that higher temperatures and higher weights also perform quite well. We also noted that very low weights (less than 0.1) essentially disable knowledge distillation, but that low temperatures ($\tau = 1$) can actually decrease the accuracy of the student. Softening the outputs of a network with an increased temperature is of utmost importance if we want knowledge distillation to work at all.

Then, we tested various students, with varying number of parameters, as well as various teachers, to determine to what extend does knowledge distillation depend on the architecture of the student of the chosen teacher. We did not notice any particular relation between the accuracy of a student, its architecture (depth and width) and the architecture of its teacher, suggesting that the performance of knowledge distillation is mostly related to the choice of hyperparameters, the capacity of the student and the chosen teacher, and less on architectural characteristics.

The choice of teacher is probably the most important factor on the success. Depending on the teacher, and on the student capacity, the student can gain as much as 6% more accuracy when trained with HKD, or barely improving at all. The best teachers are those whose training was stopped before the end of the training; other works have already noted this effect, and have attributed it to the fact that early-stopped networks behave like networks with a lower capacity. We also note that increased accuracy and teacher size is not a guarantee of higher accuracy: the smallest teachers tended to outperform larger teachers when it comes to training students. However, this may be caused by badly trained teachers; additional experiments on properly trained teachers should be performed. Finally, some teachers managed to train students that surpass them, a common phenomenon observed when performing knowledge distillation.

We also analyzed the theoretical impact of knowledge distillation on the gradients computed during the training of deep networks, and we analyzed the output of real-world models, to gain understanding about the real-life conditions in which knowledge distillation operates. Notably,

we noticed that model outputs tend to have a very stable distribution across the training, when trained with distillation, while models trained alone see the distribution of their outputs $z$ constantly shift to higher values. Knowledge distillation seems to have a normalization effect on the model output.

Overall, we can consider that knowledge distillation has proved its potential for training small networks aimed at embedded applications, although the exact factors influencing its effectiveness are not completely characterized, and there is still a need to conduct more detailed and significant works before it can be considered a go-to tool for deep learning applications.

# Future works

After so much work, a lot of questions remain about knowledge distillation. Questions about its working principles, its interpretations, its best usage, and its potential.

In distillation, the temperature is used to divide the outputs of the network before the softmax. In practice, this looks a lot like a form of normalization of the values of the logits. As the values of the logits are essentially fixed by the values of the weights and input activations in the layer before, and these weights and activations depend on their initial value at the start of the training, it could be that the optimal temperature is dependent on the distribution of logits, and that adapting it to this distribution could lead to even higher accuracy. It would be very interesting to see the effect of providing a different value for the temperature used in the student softmax and the teacher softmax, based on these distributions.

One of the proposed explanation for the workings of HKD is that it transfers information about the possible relations between class labels by transferring pseudo-probabilities about those class labels. This assumes that pseudo-probabilities are indeed representative of the level of confidence of the teacher, and that pseudo-probabilities that are close reflect the similarity of the associated labels. We think it would be extremely informative to experimentally test this hypothesis.

To determine the similarity between labels — an information that is usually not available —, we could use the confusion matrix as a proxy for measuring this: the more a class $j$ was wrongly predicted when the correct class is $i$, the closer the classes $i$ and $j$ could be. Based on this, we could examine if the distances between pseudo-probabilities indeed follow the distances computed from our confusion matrix.

Another set of experiments could leverage knowledge distillation by introducing mistakes in the output pseudo-probabilities of the teacher during distillation, by e.g; swapping the probabilities for two unrelated labels. Depending on which labels are affected (e.g. does it include the predicted label, the 2nd predicted label...), if knowledge distillation still manages to improve the student, or in the contrary it fails, we could have an idea of which relations are actually transferred or which are important.

Another explanation for HKD is that it helps the student by telling it the confidence of the teacher on its predictions: a reduced confidence from the teacher could mean that the sample is difficult to classify, and instead of impacting its accuracy on other samples to have a good accuracy for this sample, the weighted loss allows the student to place itself in an intermediate position between the certainty of the cross-entropy loss and the uncertainty of the teacher.

To test this hypothesis, we could for instance fake the outputs of the teacher for each sample of a particular class, reduce the certainty of its predictions, and present this fake output to the student. We would expect the student to have a lower confidence when classifying samples from this class. This hypothesis can also be verified by looking at the output from the student for samples misclassified or with low certainty, and see if the student provides lower pseudo-probabilities compared to what it has when trained without distillation.

In chapter 3, we analyzed the outputs of models by looking at the distribution, across the dataset, of the values of $z$ and $q$ for the predicted and target classes, and in the case of wrong predictions. It would be interesting to see what the distributions look like for the 2nd greatest output, 3rd greatest etc. It would also be useful to know the distribution of the rank of the target label when a sample is misclassified.

We only tested a very basic version of knowledge distillation. $\lambda_{HKD}$ was kept constant during training, we reused the same training setup with the same number of epochs and the same learning rate. Other works ([7]) have suggested that stopping early the knowledge distillation could have a beneficial effect. Also, the stability of output distributions could indicate that the training can be significantly shortened, which is helpful considering the computational load of an additional teacher on the training device.

At its core, knowledge distillation tries to improve the training of networks by providing additional information that is not available in the raw dataset, like relative pseudo-probabilities between classes. This may indicate that richer, more complete datasets, with much more information per sample than just the correct label, could prove to drastically improve the accuracy of deep neural networks. However, such a dataset would also be more difficult to collate, but we could leverage extremely accurate networks to fill out the blank spaces in the dataset — essentially embedding the teacher into the dataset.

Lastly, it is disturbing that we can train relatively small networks, e.g. with knowledge distillation, and reach higher accuracy than if it was trained with a regular training setup. In the end, we did not provide an augmented dataset, so all the information required to train the small network is already there when we train it alone, and as they reached a comparable level of accuracy as that or larger networks, they have the capacity to reach that level, so why do we need to train it with the help of a larger model (either as a teacher during distillation, or as the base model before pruning)? This suggests that our training setups are suboptimal, and that we can probably further reduce the size of the networks by developing new techniques.

Overall, there are still a lot of interesting questions that could be and should be answered, for knowledge distillation in particular, and for deep learning in general. Deep learning continues to be a fertile place for new ideas and new research questions.

# Appendix A

# Architectures used for the experiments

This chapter details the architectures used for the experiments. The main architecture used is based on ResNets, introduced in [17].

| Layer name | Output size | Model name | | |
|---|---|---|---|---|
| | | All other networks | ResNet34 | ResNet50 |
| conv1 | $32 \times 32$ | [3x3, $W$] | | |
| layer1 | $32 \times 32$ | $\begin{bmatrix} 3 \times 3, W \\ 3 \times 3, W \end{bmatrix} \times N_1$ | $\begin{bmatrix} 3 \times 3, W \\ 3 \times 3, W \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, W \\ 3 \times 3, W \\ 1 \times 1, 4 \cdot W \end{bmatrix} \times 3$ |
| layer2 | $16 \times 16$ | $\begin{bmatrix} 3 \times 3, 2 \cdot W \\ 3 \times 3, 2 \cdot W \end{bmatrix} \times N_2$ | $\begin{bmatrix} 3 \times 3, 2 \cdot W \\ 3 \times 3, 2 \cdot W \end{bmatrix} \times 4$ | $\begin{bmatrix} 1 \times 1, 2 \cdot W \\ 3 \times 3, 2 \cdot W \\ 1 \times 1, 2 \cdot 4 \cdot W \end{bmatrix} \times 3$ |
| layer3 | $8 \times 8$ | $\begin{bmatrix} 3 \times 3, 4 \cdot W \\ 3 \times 3, 4 \cdot W \end{bmatrix} \times N_3$ | $\begin{bmatrix} 3 \times 3, 4 \cdot W \\ 3 \times 3, 4 \cdot W \end{bmatrix} \times 6$ | $\begin{bmatrix} 1 \times 1, 4 \cdot W \\ 3 \times 3, 4 \cdot W \\ 1 \times 1, 4 \cdot 4 \cdot W \end{bmatrix} \times 6$ |
| layer4 | $4 \times 4$ | - | $\begin{bmatrix} 3 \times 3, 8 \cdot W \\ 3 \times 3, 8 \cdot W \end{bmatrix} \times 3$ | $\begin{bmatrix} 1 \times 1, 8 \cdot W \\ 3 \times 3, 8 \cdot W \\ 1 \times 1, 8 \cdot 4 \cdot W \end{bmatrix} \times 3$ |
| | $1 \times 1$ | average pool | | |
| | $C$ | fully connected | | |
| | $C$ | softmax | | |

Table A.1: Architectures of the various networks. $W$ denotes the base width, $C$ the number of target classes. $N_i$ are additional architecture parameters given in table A.2.

Table A.1 gives the overall structure of the networks. All convolutional layers are followed by a batch normalization layer. With the exception of the softmax at the end, the only nonlinearity used is the Rectified Linear Unit, ReLU [38].

| Model name | $N_1$ | $N_2$ | $N_3$ |
|---|---|---|---|
| ResNet20 | 3 | 3 | 3 |
| ResNet32 | 5 | 5 | 5 |
| ResNet44 | 7 | 7 | 7 |
| ResNet56 | 9 | 9 | 9 |

Table A.2: Additional parameters for the networks designed for CIFAR.

The networks contain a sequence of either basic *residual blocks* or of *bottleneck blocks* (only for ResNet50). Basic blocks are composed of two stacked convolutional layers (and their batch normalization layer) separated by ReLU nonlinearity. Bottleneck blocks consist of three stacked convolutional layers, with ReLU in between, the middle one is a normal $3 \times 3$, the outer two perform $1 \times 1$ convolutions to reduce the number of channels at the input of the $3 \times 3$ convolution, hence the name. In both case, the input of the block is added to the output of this stack of convolutional layers, and is then passed through another ReLU nonlinearity: the path that goes through the stack of layers is the residual path, the other is called the shortcut path.

Similarly to Wide ResNets [63], the number of channels per layer can be scaled by changing the value of the parameter $W$, the base width. The default, as used on CIFAR-10, is $W = 16$.

The first $3 \times 3$ convolutional layer of the first residual block of layer2, layer3 and layer4 has a stride of 2, in order to halve the size of its output feature map. It also outputs twice as much channels as it received in input. The shortcut connection also has a $1 \times 1$ convolution with a stride of 2, to ensure that the feature maps have the right dimensions when adding them.

# Appendix B

# Additional results

This appendix gives additional results of the experiments.

## B.1 Distillation of students with varying teachers

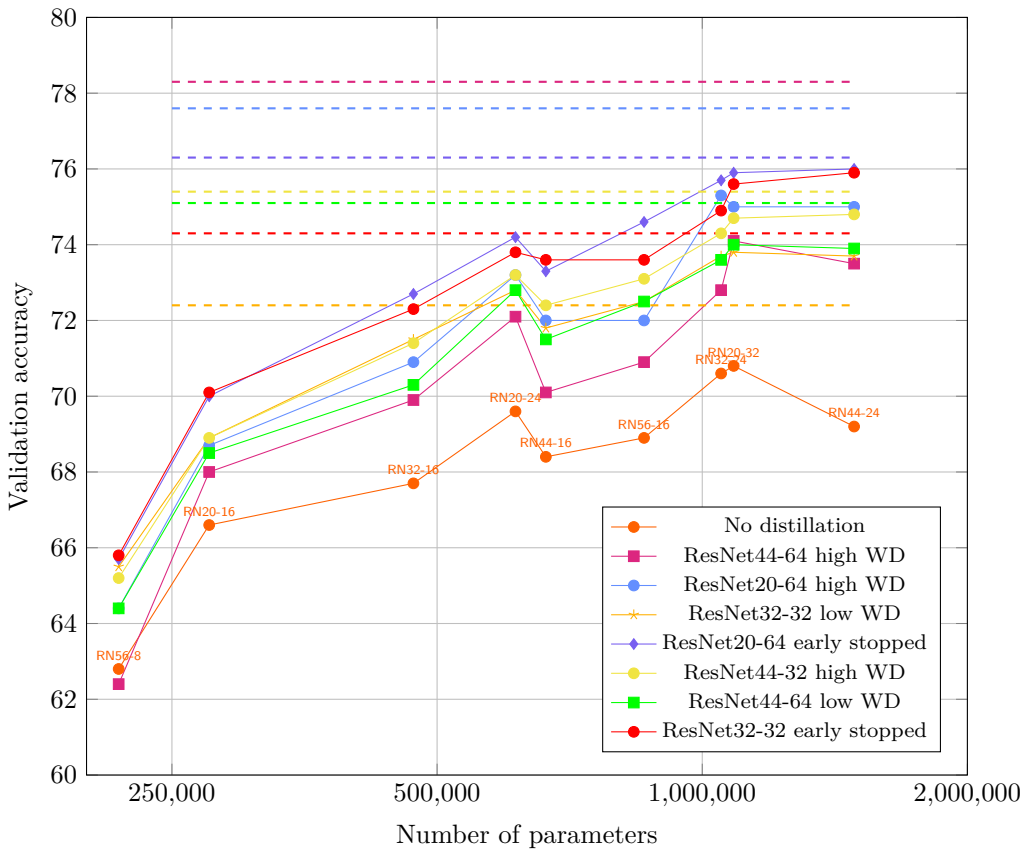



Figure B.1: Distillation with students of various capacity, and with different teachers. All students are trained with initial learning rate 0.1 and low weight decay. The dashed lines represent the accuracy of the teachers.

# Appendix C

# Computation of the gradients of the losses

Computing the gradients of the various losses involved may look innocuous, however some mathematical technicalities can arrive during their derivation.

## C.1 The softmax function

The softmax function takes a vector of logits, $z \in \mathbb{R}^C$, and outputs a vector of class "probabilities", $q \in \mathbb{R}^C$. Its definition is

$$\operatorname{softmax}(z)[i] = \frac{\exp(z[i])}{\sum_k \exp(z[k])}. \tag{C.1}$$

We also have $\sum_i \operatorname{softmax}(z)[i] = 1$, as they represent probabilities.

We can compute the partial derivatives easily (with $\delta_{ij}$ being the Kronecker delta):

$$\begin{aligned}
\frac{\partial \operatorname{softmax}(z)[i]}{\partial z[j]} &= \frac{\frac{\partial}{\partial z[j]}\big(\exp(z[i])\big) \cdot \sum_k \big(\exp(z[k])\big) - \exp(z[i]) \cdot \frac{\partial}{\partial z[j]} \sum_k \big(\exp(z[k])\big)}{\left(\sum_k \exp(z[k])\right)^2} \\
&= \frac{\delta_{ij} \exp(z[i]) \sum_k \big(\exp(z[k])\big) - \exp(z[i]) \cdot \sum_k \left(\frac{\partial}{\partial z[j]} \exp(z[k])\right)}{\left(\sum_k \exp(z[k])\right)^2} \\
&= \frac{\delta_{ij} \exp(z[i]) \sum_k \big(\exp(z[k])\big) - \exp(z[i]) \exp(z[j])}{\left(\sum_k \exp(z[k])\right)^2} \\
&= \frac{\exp(z[i])}{\sum_k \exp(z[k])} \cdot \left(\delta_{ij} - \frac{\exp(z[j])}{\sum_k \exp(z[k])}\right) \\
&= \operatorname{softmax}(z)[i] \cdot \big(\delta_{ij} - \operatorname{softmax}(z)[j]\big).
\end{aligned} \tag{C.2}$$

In some situations, one is interested in a different definition of the softmax, which includes the "temperature" $\tau$ used in knowledge distillation:

$$\operatorname{softmax}(z;\tau)[i] = \frac{\exp\big(z[i]/\tau\big)}{\sum_k \exp\big(z[k]/\tau\big)}. \tag{C.3}$$

It is easy to show that the partial derivatives are then:

$$\frac{\partial \operatorname{softmax}(z;\tau)[i]}{\partial z[j]} = \frac{1}{\tau} \operatorname{softmax}(z;\tau)[i] \cdot (\delta_{ij} - \operatorname{softmax}(z;\tau)[j]) \tag{C.4}$$

For the rest of this section, we will use a shorthand notation for the softmax function:

$$\begin{aligned}
S(z) &= \operatorname{softmax}(z) \\
S(z;\tau) &= \operatorname{softmax}(z;\tau).
\end{aligned}$$

In addition, when $z$ is a vector of logits, the output of the softmax can be regarded as a vector of "probabilities" on which the cross-entropy losses act upon. We note $q[i] = \text{softmax}(z)[i]$ these outputs as class probabilities.

### C.1.1 Properties of the softmax

First, we can note that the softmax function doesn't change when a constant is added to each component of its input:

$$\begin{aligned}\text{softmax}(a+t)[i] &= \frac{\exp(a[i]+t)}{\sum_j \exp(a[j]+t)} \\ &= \frac{\exp(t)\exp(a[i])}{\sum_j \exp(t)\exp(a[j])} \\ &= \frac{\exp(a[i])}{\sum_j \exp(a[j])} \\ &= \text{softmax}(a)[i].\end{aligned}$$

Notably, this means that we can regularize the logits so that they have a zero mean, i.e. $\sum_i a[i] = 0$. Alternatively, we can let the network normalize its logits (e.g. due to batch normalization or an $l_2$ norm), and only care about the relative difference between the digits. However, we still have to care about the *scale* of this relative difference on the output of the softmax: a smaller relative difference between the output components will lead to a lower saturation of the softmax output, a property that is used in knowledge distillation by increasing the so-called temperature.

## C.2 The standard cross-entropy loss

The standard cross-entropy loss for a sample $x^{(t)}$ whose logits are $z^{(t)}$ is:

$$\text{loss}_{CE}^{(t)} = -\log\Big(\text{softmax}(z^{(t)})[y^{(t)}]\Big) \tag{C.5}$$

where $y^{(t)}$ is the sample's "true" class. We want to stress that this is the value of the loss for this particular sample $x^{(t)}$: the loss is aggregated (summed) over all the samples from the mini-batch $\mathcal{B}$.

The gradient of the loss relative to the logits can be computed as:

$$\begin{aligned}\frac{\partial\,\text{loss}_{CE}^{(t)}}{\partial\, z^{(t)}[j]} &= \frac{\partial}{\partial\, z^{(t)}[j]}\Big(-\log\big(S(z^{(t)})[y^{(t)}]\big)\Big) \\ &= -\frac{1}{S(z)[y]}\frac{\partial}{\partial\, z[j]}\big(S(z)[y^{(t)}]\big) \\ &= -\frac{1}{S(z)[y_{\text{target}}]}S(z)[y_{\text{target}}]\big(\delta_{y_{\text{target}},j} - S(z)[j]\big) \\ &= \text{softmax}(z^{(t)})[j] - \delta_{y^{(t)},j}.\end{aligned} \tag{C.6}$$

Again, the gradient is aggregated over all samples in the mini-batch. In the following, we will leave out the exponents $\cdot^{(t)}$.

## C.3 The knowledge distillation loss

The knowledge distillation loss, as proposed in [22], for a sample $x$ whose logits produced by the student model are $z_S$, and whose logits produced by the teacher model are $z_T$, is:

$$\text{loss}_{HKD} = -\tau^2 \sum_i S(z_T;\tau)[i]\log(S(z_S;\tau)[i]). \tag{C.7}$$

The gradient of the loss relative to the logits of the student is:

$$\begin{aligned}
\frac{\partial \operatorname{loss}_{HKD}}{\partial z_S[j]} &= -\tau^2 \sum_i \left( S(z_T;\tau)[i] \frac{\partial}{\partial z_S[j]} \log\big(S(z_S;\tau)[i]\big) \right) \\
&= -\tau^2 \sum_i \left( S(z_T;\tau)[i] \frac{1}{S(z_S;\tau)[i]} \frac{\partial}{\partial z_S[j]} S(z_S;\tau)[i] \right) \\
&= -\tau^2 \sum_i \left( S(z_T;\tau)[i] \frac{1}{S(z_S;\tau)[i]} \frac{1}{\tau} S(z_S;\tau)[i](\delta_{ij} - S(z_S;\tau)[j]) \right) \\
&= -\tau \sum_i \Big( S(z_T;\tau)[i](\delta_{ij} - S(z_S;\tau)[j]) \Big) \\
&= -\tau \Big( \sum_i \big(S(z_T;\tau)[i]\delta_{ij}\big) - \sum_i \big(S(z;\tau)[i] S(z_S;\tau)[j]\big) \Big) \\
&= -\tau \Big( S(z_T;\tau)[j] - S(z_S;\tau)[j]) \sum_i \big(S(z;\tau)[i]\big) \Big) \\
&= -\tau \big( S(z_T;\tau)[j] - S(z_S;\tau)[j] \big) \\
&= \tau (S(z_S;\tau)[j] - S(z_T;\tau)[j]). \qquad \text{(C.8)}
\end{aligned}$$

It makes no sense to compute the gradient of the loss relative to the logits of the teacher, as they are fixed during the training of the student.

### C.3.1 Limiting behavior of the gradient when $\tau \to \infty$

The original definition of the distillation loss by [22] scaled the distillation loss by the square of the temperature. This was done so that[1]

$$\begin{aligned}
\lim_{\tau\to\infty} \frac{\partial \operatorname{loss}_{HKD}}{\partial z_S[j]} &= \lim_{\tau\to\infty} \tau \left( \frac{1}{\sum_k \exp\left(\frac{z_S[k]-z_S[j]}{\tau}\right)} - \frac{1}{\sum_k \exp\left(\frac{z_T[k]-z_S[j]}{\tau}\right)} \right) \\
&= \lim_{\tau\to\infty} \tau \frac{\sum_k \left(\exp\left(z_T[k]/\tau\right)\right) - \sum_k \left(\exp\left(z_S[k]/\tau\right)\right)}{\left(\sum_k \exp\left(\frac{z_S[k]-z_S[j]}{\tau}\right)\right)\left(\sum_k \exp\left(\frac{z_T[k]-z_T[j]}{\tau}\right)\right)} \\
&= \lim_{\tau\to\infty} \tau \frac{\sum_k \left(1 + \frac{z_T[k]-z_T[j]}{\tau} + \mathcal{O}\left(\breve{z_T}^2\right) - 1 - \frac{z_S[k]-z_S[j]}{\tau} - \mathcal{O}\left(\breve{z_S}^2\right)\right)}{\left(\sum_k 1 + \frac{z_S[k]-z_S[j]}{\tau} + \mathcal{O}\left(\breve{z_S}^2\right)\right)\left(\sum_k 1 + \frac{z_T[k]-z_T[j]}{\tau} + \mathcal{O}\left(\breve{z_T}^2\right)\right)} \\
&= \frac{\sum_k z_T[k] - z_T[j] - z_S[k] + z_S[j]}{\left(\sum_k 1\right)\left(\sum_k 1\right)} \\
&= \frac{C \cdot z_S[j] - C \cdot z_T[j] + \sum_k z_T[k] - \sum_k z_S[k]}{C^2} \\
&= \frac{1}{C}(z_S[j] - z_T[j] + \frac{1}{C}\sum_k z_T[k] - \frac{1}{C}\sum_k z_S[k]) \\
&= \frac{1}{C}(z_S[j] - z_T[j] - (\overline{z_S} - \overline{z_T})) = \frac{1}{C}\big((z_S[j] - \overline{z_S}) - (z_T[j] - \overline{z_T})\big) \qquad \text{(C.9)}
\end{aligned}$$

matches the form of the gradient of the mean square error

$$\operatorname{loss}_{MSE} = \frac{1}{2}\frac{1}{C}\sum_i (z_S[i] - z_T[i])^2 \qquad \text{(C.10)}$$

$$\frac{\partial \operatorname{loss}_{MSE}}{\partial z_S[j]} = \frac{1}{C}(z_S[j] - z_T[j]), \qquad \text{(C.11)}$$

[1] $\overline{z_S} = \frac{1}{C}\sum_k z_S[k]$ denotes the mean of the logits $z_S$, and similarly for $\overline{z_T}$. If the logits are centered, these terms disappear. $\breve{z_S}$ denotes the expression $\frac{z_S[k]-z_S[j]}{\tau}$; this is to save space.

with the $\overline{z_S}$ and $\overline{z_T}$ variables being the only differences.

### C.3.2 The gradient relative to the class probabilities

Things get interesting when computing the gradient of the loss relative to the class probabilities $q^{(t)}[j]$. A first technique would be to apply standard rules of partial derivatives:

$$\frac{\partial \, \text{loss}_{HKD}}{\partial \, q_S[j]} = \frac{\partial}{\partial \, q_S[j]} \Big( - \tau^2 \sum_i \big( q_T[i] \log(q_S[i]) \big) \Big) \tag{C.12}$$

$$= -\tau^2 \sum_i \left( q_T[i] \frac{\partial}{\partial \, q_S[j]} \log(q_S[i]) \right) \tag{C.13}$$

$$= -\tau^2 \sum_i \left( q_T[i] \frac{1}{q_S[i]} \frac{\partial \, q_S[i]}{\partial \, q_S[j]} \right) \tag{C.14}$$

$$= -\tau^2 \sum_i \left( q_T[i] \frac{1}{q_S[i]} \delta_{ij} \right) \tag{C.15}$$

$$= -\tau^2 \frac{q_T[j]}{q_S[j]}. \tag{C.16}$$

However, this expression is not correct. Indeed, the gradient of a function should be zero at points corresponding to extrema of the function. Here, we know, from the expression of the partial derivative w.r.t. the logits, that the gradient is zero when $q_S[j] = q_T[j]$, which is confirmed by the intuition: the two distributions compared by the cross-entropy are then equal, and so the cross-entropy is at a minimum. However, this expression of the partial derivatives w.r.t. the class probabilities does not respect this property, and admits a zero only if $q_T[j] = 0 \; \forall j$.

The expression is incorrect because we have wrongly assumed that $\frac{\partial \, q_S[i]}{\partial \, q_S[j]} = \delta_{ij}$, which is false: due to the properties of the softmax, the $C$ variables $q_S[j]$ are dependent on the value of the $C-1$ others, so the partial derivative relative to one variable should specify which $C-2$ variables are kept constant and independent, and thus which remaining variable is allowed to vary to absorb the changes in the first variable.

Curiously, if we inject equation C.16 into the formula of the chain rule, in order to compute the gradient relative to the student logits, everything seems fine:

$$\frac{\partial \, \text{loss}_{HKD}}{\partial \, z_S[j]} = \sum_k \frac{\partial \, \text{loss}_{HKD}}{\partial \, q_S[k]} \frac{\partial \, q_S[k]}{\partial \, z_S[j]} \tag{C.17}$$

$$= -\tau^2 \sum_k \left( \frac{q_T[k]}{q_S[k]} \cdot \frac{1}{\tau} q_S[k] \big( \delta_{kj} - q_S[j] \big) \right) \tag{C.18}$$

$$= -\tau \sum_k \Big( q_T[k] \big( \delta_{kj} - q_S[j] \big) \Big) \tag{C.19}$$

$$= -\tau \Big( \sum_k \big( q_T[k] \delta_{kj} \big) - \sum_k \big( q_T[k] q_S[j] \big) \Big) \tag{C.20}$$

$$= -\tau \big( q_T[j] - q_S[j] \big) \tag{C.21}$$

$$= \tau \big( q_S[j] - q_T[j] \big), \tag{C.22}$$

the last 4 lines being extremely similar to the last lines of the correct derivation of $\frac{\partial \, \text{loss}_{HKD}}{\partial \, z_S[j]}$ (equation (C.8)). This is because this computation is disregarding the fact that one component of the $q_S$ variable is dependent on the other ones, and thus it shouldn't appear in this summation, its effect having been handled by the other variables. However, as we are using the incorrect partial derivatives, and are summing across all the variables, this error cancels out and gives the correct answer.

This computation is exactly what PyTorch's AutoGrad mechanism is doing. Which, as we can see with (C.22), works out sufficiently fine when training a neural network with it. As we are

usually worried about the derivatives w.r.t. the raw outputs of the model, and not w.r.t. the pseudo-probabilities, this is not usually a problem.

A correct derivation of the gradient should consider that $\frac{\partial\, q_S[i]}{\partial\, q_S[j]} \neq \delta_{ij}$. For instance, for the simple case of $C = 2$, $\frac{\partial\, q_S[i]}{\partial\, q_S[j]} = +1$ if $i = j$, and $-1$ otherwise:

$$\frac{\partial\, \text{loss}_{HKD}}{\partial\, q_S[j]} = -\tau^2 \sum_i \left( q_T[i] \frac{1}{q_S[i]} \frac{\partial\, q_S[i]}{\partial\, q_S[j]} \right) \qquad \text{Restarting from (C.14)} \tag{C.23}$$

$$= -\tau^2 \sum_i \left( q_T[i] \frac{1}{q_S[i]} (-1) \cdot (-1)^{\delta_{ij}} \right) \tag{C.24}$$

$$= -\tau^2 \left( \frac{q_T[j]}{q_S[j]} - \frac{q_T[2-j]}{q_S[2-j]} \right) \tag{C.25}$$

$$= -\tau^2 \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) \tag{C.26}$$

where we have used the fact that we have only 2 classes, so $q_S[1] + q_S[2] = 1$ (in 1-based indexing). This expression is much more correct, as it admits zeroes when $\frac{q_T[j]}{q_S[j]} = \frac{1-q_T[j]}{1-q_S[j]}$, which happens right when $q_T[j] = q_S[j]$. In addition, numerical explorations confirm that this expression indeed gives the correct values for the gradient.

However, care must be taken if we use this derivative with the chain rule to obtain $\frac{\partial\, \text{loss}_{HKD}}{\partial\, z_S[j]}$: naively reinjecting (C.26) into the chain rule and still summing over both components leads to a strange result:

$$\frac{\partial\, \text{loss}_{HKD}}{\partial\, z_S[j]} = \sum_k \frac{\partial\, \text{loss}_{HKD}}{\partial\, q_S[k]} \frac{\partial\, q_S[k]}{\partial\, z_S[j]} \tag{C.27}$$

$$= -\tau^2 \sum_k \left[ \left( \frac{q_T[k]}{q_S[k]} - \frac{1 - q_T[k]}{1 - q_S[k]} \right) \left( \tfrac{1}{\tau} q_S[k](\delta_{kj} - q_S[j]) \right) \right] \tag{C.28}$$

$$\begin{aligned} = -\tau \Bigg[ & \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) q_S[j](1 - q_S[j]) \\ & + \left( \frac{q_T[2-j]}{q_S[2-j]} - \frac{1 - q_T[2-j]}{1 - q_S[2-j]} \right) q_S[2-j](\delta_{2-j,j} - q_S[j]) \Bigg] \end{aligned} \tag{C.29}$$

$$\begin{aligned} = -\tau \Bigg[ & \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) q_S[j](1 - q_S[j]) \\ & + \left( \frac{1 - q_T[j]}{1 - q_S[j]} - \frac{q_T[j]}{q_S[j]} \right) (1 - q_S[j])(-q_S[j]) \Bigg] \end{aligned} \tag{C.30}$$

$$\begin{aligned} = -\tau \Bigg[ & \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) q_S[j](1 - q_S[j]) \\ & + \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) q_S[j](1 - q_S[j]) \Bigg] \end{aligned} \tag{C.31}$$

$$= -\tau \cdot 2 \cdot \left( \frac{q_T[j]}{q_S[j]} - \frac{1 - q_T[j]}{1 - q_S[j]} \right) q_S[j](1 - q_S[j]) \tag{C.32}$$

$$= -\tau \cdot 2 \cdot \big( q_T[j](1 - q_S[j]) - q_S[j](1 - q_T[j]) \big) \tag{C.33}$$

$$= -\tau \cdot 2 \cdot \big( q_T[j] - q_T[j] q_S[j] - q_S[j] + q_S[j] q_T[j] \big) \tag{C.34}$$

$$= 2 \cdot \tau \cdot (q_S[j] - q_T[j])). \tag{C.35}$$

which has an additional factor of 2. Indeed, we need to take only one component into account, as the other one is dependent.

## C.4 The gradient of the cross-entropy loss relative to the class probabilities

Similarly to the case of the HKD loss, the gradient for the standard cross-entropy loss is not trivially computed. The naive technique, which ignores the relation between $p[1]$ and $p[2]$, gives:

$$\frac{\partial\,\text{loss}_{CE}}{\partial\,q[j]} = \frac{\partial}{\partial\,q[j]}\Big(-\log\big(\underbrace{S(z)[y_{\text{target}}]}_{q[y_{\text{target}}]}\big)\Big) \tag{C.36}$$

$$= -\frac{1}{q[y_{\text{target}}]}\frac{\partial}{\partial\,q[j]}\big(q[y_{\text{target}}]\big) \tag{C.37}$$

$$= -\frac{1}{q[y_{\text{target}}]}\delta_{y_{\text{target}},j}. \tag{C.38}$$

The gradient is nonzero only in the direction of the target class.

However, the correct expression is:

$$\frac{\partial\,\text{loss}_{CE}}{\partial\,q[j]} = -\frac{1}{q[y_{\text{target}}]}\frac{\partial\,q[y_{\text{target}}]}{\partial\,p[j]} \tag{C.39}$$

$$= -\frac{1}{q[y_{\text{target}}]}\cdot\begin{cases}1 & \text{if } y_{\text{target}} = j\\ -1 & \text{otherwise}\end{cases} \tag{C.40}$$

$$= \begin{cases}-\frac{1}{q[y_{\text{target}}]} & \text{if } y_{\text{target}} = j\\ +\frac{1}{q[y_{\text{target}}]} & \text{otherwise}\end{cases}. \tag{C.41}$$

The same curiosity that was observed with the HKD loss gradient can be observed with the cross-entropy loss gradient: if we inject (C.38) into the chain rule to compute $\frac{\partial\,\text{loss}_{CE}}{\partial\,z[j]}$, we have:

$$\frac{\partial\,\text{loss}_{CE}}{\partial\,z[j]} = \sum_k \frac{\partial\,\text{loss}_{CE}}{\partial\,q[k]}\frac{\partial\,q[k]}{\partial\,z[j]} \tag{C.42}$$

$$= \sum_k -\frac{1}{q[y_{\text{target}}]}\delta_{y_{\text{target}},k}q[k](\delta_{kj} - q[j]) \tag{C.43}$$

$$= -\frac{1}{q[y_{\text{target}}]}q[y_{\text{target}}](\delta_{y_{\text{target}},j} - q[j]) \tag{C.44}$$

$$= q[j] - \delta_{y_{\text{target}},j}, \tag{C.45}$$

which is the correct expression as in (C.6).

However, if we inject the correct gradient given in (C.41) into the chain rule, we obtain a much different result:

$$\frac{\partial\,\text{loss}_{CE}}{\partial\,z[j]} = \sum_k \frac{\partial\,\text{loss}_{CE}}{\partial\,q[k]}\frac{\partial\,q[k]}{\partial\,z[j]} \tag{C.46}$$

$$\begin{aligned} = & -\frac{1}{q[y_{\text{target}}]}q[y_{\text{target}}](\delta_{y_{\text{target}},j} - q[j])\\ & + \frac{1}{q[y_{\text{target}}]}q[2 - y_{\text{target}}](\delta_{(2-y_{\text{target}}),j} - q[j]).\end{aligned} \tag{C.47}$$

If $j = y_{\text{target}}$, then we have:

$$\frac{\partial\,\text{loss}_{CE}}{\partial\,z[j]} = -(1 - q[j]) + \frac{1}{q[y_{\text{target}}]}(1 - q[y_{\text{target}}])(0 - q[y_{\text{target}}]) \tag{C.48}$$

$$= q[j]) - 1 + q[y_{\text{target}}] - 1 \tag{C.49}$$

$$= 2(q[j] - 1). \tag{C.50}$$

If $j \neq y_{\text{target}}$, then we have:

$$\frac{\partial \, \text{loss}_{CE}}{\partial \, z[j]} = -(0 - q[j]) + \frac{1}{q[y_{\text{target}}]}(1 - q[y_{\text{target}}])(1 - q[j]) \tag{C.51}$$

$$= q[j] + \frac{1}{q[y_{\text{target}}]}(q[j])q[y_{\text{target}}] \tag{C.52}$$

$$= q[j] + q[j] = 2q[j]. \tag{C.53}$$

Both cases can be combined as:

$$\frac{\partial \, \text{loss}_{CE}}{\partial \, z[j]} = 2 \cdot (q[j] - \delta_{y_{\text{target}},j}), \tag{C.54}$$

again with an additional factor of 2 compared to the expected expression, which disappears if we only take one term, as we should.

# Bibliography


[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, *TensorFlow: Large-scale machine learning on heterogeneous systems*, Software available from tensorflow.org, 2015. [Online]. Available: `https://www.tensorflow.org/`.

[2] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," *2017 International Joint Conference on Neural Networks (IJCNN)*, pp. 2547–2554, 2017. DOI: `10.1109/IJCNN.2017.7966166`. arXiv: `1609.00222 [cs.LG]`.

[3] Z. Allen-Zhu, Y. Li, and Z. Song, "A convergence theory for deep learning via over-parameterization," in *Proceedings of the 36th International Conference on Machine Learning*, vol. 97, 2019, pp. 242–252. [Online]. Available: `https://proceedings.mlr.press/v97/allen-zhu19a.html`.

[4] J. Ba and R. Caruana, "Do deep nets really need to be deep?" In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14, Montreal, Canada: MIT Press, 2014, pp. 2654–2662. arXiv: `1312.6184 [cs.LG]`. [Online]. Available: `https://proceedings.neurips.cc/paper/2014/file/ea8fcd92d59581717e06eb187f10666d-Paper.pdf`.

[5] C. Bucilă, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, ser. KDD '06, Philadelphia, PA, USA: Association for Computing Machinery, 2006, pp. 535–541, ISBN: 1595933395. DOI: `10.1145/1150402.1150464`. [Online]. Available: `https://doi.org/10.1145/1150402.1150464`.

[6] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, "Learning efficient object detection models with knowledge distillation," in *Advances in Neural Information Processing Systems*, vol. 30, Curran Associates, Inc., 2017. [Online]. Available: `https://proceedings.neurips.cc/paper/2017/file/e1e32e235eee1f970470a3a6658dfdd5-Paper.pdf`.

[7] J. H. Cho and B. Hariharan, "On the efficacy of knowledge distillation," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 4793–4801, 2019. DOI: `10.1109/iccv.2019.00489`. arXiv: `1910.01348 [cs.LG]`. [Online]. Available: `http://dx.doi.org/10.1109/ICCV.2019.00489`.

[8] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, *Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1*, 2016. arXiv: `1602.02830 [cs.LG]`.

[9] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *2009 IEEE Conference on computer vision and pattern recognition (CVPR)*, 2009, pp. 248–255.

[10] L. Deng, P. Jiao, J. Pei, Z. Wu, and G. Li, *Gxnor-net: Training deep neural networks with ternary weights and activations without full-precision memory under a unified discretization framework*, 2018. arXiv: 1705.09283 [cs.LG].

[11] S. S. Du, X. Zhai, B. Poczos, and A. Singh, "Gradient descent provably optimizes over-parameterized neural networks," in *International Conference on Learning Representations*, 2019. [Online]. Available: https://openreview.net/forum?id=S1eK3i09YQ.

[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: http://www.deeplearningbook.org.

[13] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, and K. He, "Accurate, large minibatch sgd: Training imagenet in 1 hour," 2017. arXiv: 1706.02677 [cs.CV].

[14] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," in *International Conference on Machine Learning*, PMLR, 2017, pp. 1321–1330. arXiv: 1706.04599 [cs.LG].

[15] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "[dl] a survey of fpga-based neural network inference accelerators," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 12, no. 1, Mar. 2019, ISSN: 1936-7406. DOI: 10.1145/3289185.

[16] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural networks," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, 2015, pp. 1135–1143. arXiv: 1506.02626 [cs.NE].

[17] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385 [cs.CV]. [Online]. Available: http://dx.doi.org/10.1109/cvpr.2016.90.

[18] ——, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034. arXiv: 1502.01852 [cs.CV].

[19] ——, "Identity mappings in deep residual networks," in *European conference on computer vision*, Springer, 2016, pp. 630–645. arXiv: 1603.05027 [cs.CV].

[20] T. He, Z. Zhang, H. Zhang, Z. Zhang, J. Xie, and M. Li, "Bag of tricks for image classification with convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 558–567. DOI: 10.1109/CVPR.2019.00065. arXiv: 1812.01187 [cs.CV].

[21] Y. He, X. Zhang, and J. Sun, "Channel pruning for accelerating very deep neural networks," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1398–1406. DOI: 10.1109/ICCV.2017.155. arXiv: 1707.06168 [cs.CV].

[22] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *NIPS Deep Learning and Representation Learning Workshop*, 2014. arXiv: 1503.02531 [stat.ML].

[23] E. Hoffer, R. Banner, I. Golan, and D. Soudry, "Norm matters: Efficient and accurate normalization schemes in deep networks," in *Advances in Neural Information Processing Systems*, vol. 31, Curran Associates, Inc., 2018. arXiv: 1803.01814 [stat.ML]. [Online]. Available: https://proceedings.neurips.cc/paper/2018/file/a0160709701140704575d499c997b6ca-Paper.pdf.

[24] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[25] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, *Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size*, 2016. arXiv: 1602.07360 [cs.CV].

[26] Y. Idelbayev, *Proper ResNet implementation for CIFAR10/CIFAR100 in PyTorch*, https://github.com/akamaster/pytorch_resnet_cifar10, Accessed: September 1st 2020.

[27] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International conference on machine learning*, ser. ICML'15, vol. 37, PMLR, 2015, pp. 448–456. arXiv: 1502.03167 [cs.LG].

[28] A. Krizhevsky, "Learning multiple layers of features from tiny images," Tech. Rep., 2009, For the CIFAR-10 and CIFAR-100 datasets see https://www.cs.toronto.edu/~kriz/cifar.html. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

[29] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Advances in neural information processing systems*, vol. 25, pp. 1097–1105, 2012.

[30] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, *Pruning filters for efficient convnets*, 2017. arXiv: 1608.08710 [cs.CV].

[31] H. Li, Z. Xu, G. Taylor, C. Studer, and T. Goldstein, "Visualizing the loss landscape of neural nets," in *Advances in Neural Information Processing Systems*, vol. 31, 2018. arXiv: 1712.09913 [cs.LG].

[32] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, "Fp-bnn: Binarized neural network on fpga," *Neurocomputing*, vol. 275, pp. 1072–1086, 2018, ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2017.09.046.

[33] Z. Liu, J. Li, Z. Shen, G. Huang, S. Yan, and C. Zhang, "Learning efficient convolutional networks through network slimming," *2017 IEEE International Conference on Computer Vision (ICCV)*, pp. 2755–2763, 2017. DOI: 10.1109/ICCV.2017.298. arXiv: 1708.06519 [cs.CV].

[34] Z. Liu, M. Sun, T. Zhou, G. Huang, and T. Darrell, "Rethinking the value of network pruning," in *International Conference on Learning Representations*, 2019. arXiv: 1810.05270 [cs.LG]. [Online]. Available: https://openreview.net/forum?id=rJlnB3C5Ym.

[35] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, *Shufflenet v2: Practical guidelines for efficient cnn architecture design*, 2018. arXiv: 1807.11164 [cs.CV].

[36] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, "Optimizing the convolution operation to accelerate deep neural networks on fpga," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 26, no. 7, pp. 1354–1367, 2018. DOI: 10.1109/TVLSI.2018.2815603.

[37] A. K. Mishra and D. Marr, *Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy*, 2018. arXiv: 1711.05852 [cs.LG].

[38] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning (ICML)*, 2010, pp. 807–814, ISBN: 9781605589077.

[39] W. Park, D. Kim, Y. Lu, and M. Cho, "Relational knowledge distillation," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 3962–3971. DOI: 10.1109/CVPR.2019.00409. arXiv: 1904.05068 [cs.CV].

[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[41] A. Polino, R. Pascanu, and D. Alistarh, "Model compression via distillation and quantization," in *International Conference on Learning Representations*, 2018. arXiv: 1802.05668 [cs.NE]. [Online]. Available: https://openreview.net/forum?id=S1XolQbRW.

[42] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, "Fitnets: Hints for thin deep nets," vol. abs/1412.6550, 2015. arXiv: 1412.6550 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1412.6550.

[43] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015. DOI: 10.1007/s11263-015-0816-y.

[44] M. Sandler, A. G. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018. DOI: 10.1109/CVPR.2018.00474. arXiv: 1801.04381 [cs.CV].

[45] S. Santurkar, D. Tsipras, A. Ilyas, and A. Mądry, "How does batch normalization help optimization?" In *Proceedings of the 32nd international conference on neural information processing systems*, 2018, pp. 2488–2498. arXiv: 1805.11604 [stat.ML].

[46] F. de Saussure, *Course in general linguistics*, trans. by R. Harris. London: Duckworth, 1983, ISBN: 0-8126-9023-0.

[47] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019. DOI: 10.1109/ACCESS.2018.2890150.

[48] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2015. arXiv: 1409.1556 [cs.CV].

[49] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-first AAAI conference on artificial intelligence*, 2017. arXiv: 1602.07261 [cs.CV].

[50] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594. arXiv: 1409.4842 [cs.CV].

[51] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826. arXiv: 1512.00567 [cs.CV].

[52] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, *Mnasnet: Platform-aware neural architecture search for mobile*, 2019. arXiv: 1807.11626 [cs.CV].

[53] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, PMLR, 2019, pp. 6105–6114. arXiv: 1905.11946 [cs.LG]. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html.

[54] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2017. DOI: 10.1145/3020078.3021744. [Online]. Available: http://dx.doi.org/10.1145/3020078.3021744.

[55] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: Mapping Regular and Irregular Convolutional Neural Networks on FPGAs," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–17, 2018, ISSN: 2162-237X. DOI: 10.1109/TNNLS.2018.2844093.

[56] ——, "Latency-Driven Design for FPGA-based Convolutional Neural Networks," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Sep. 2017, pp. 1–8. DOI: 10.23919/FPL.2017.8056828.

[57] S. I. Venieris and C.-S. Bouganis, "Fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 40–47. DOI: 10.1109/FCCM.2016.22.

[58] E. Wang, J. J. Davis, P. Y. K. Cheung, and G. A. Constantinides, "LUTNet: Rethinking inference in FPGA soft logic," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2019.

[59] E. Wang, J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Cheung, and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going," *ACM Computing Surveys (CSUR)*, vol. 52, no. 2, pp. 1–39, May 2019. DOI: 10.1145/3309551. arXiv: 1901.06955 [cs.CV].

[60] S. Xie, R. B. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," 2017, pp. 5987–5995. arXiv: 1611.05431 [cs.CV].

[61] M. Ye, C. Gong, L. Nie, D. Zhou, A. Klivans, and Q. Liu, "Good subnetworks provably exist: Pruning via greedy forward selection," in *Proceedings of the 37th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, vol. 119, 2020, pp. 10 820–10 830. arXiv: 2003.01794 [cs.LG]. [Online]. Available: https://proceedings.mlr.press/v119/ye20b.html.

[62] S. Zagoruyko and N. Komodakis, "Paying more attention to attention: Improving the performance of convolutional neural networks via attention transfer," in *ICLR*, vol. abs/1612.03928, 2017. arXiv: 1612.03928 [cs.CV]. [Online]. Available: https://openreview.net/forum?id=Sks9_ajex.

[63] ——, *Wide residual networks*, 2016. arXiv: 1605.07146 [cs.CV].

[64] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15, Association for Computing Machinery, 2015, pp. 161–170, ISBN: 9781450333153. DOI: 10.1145/2684746.2689060.

[65] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," *ArXiv*, vol. abs/1611.03530, 2017, A newer version was published as [66].

[66] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning (still) requires rethinking generalization," vol. 64, no. 3, pp. 107–115, 2021, ISSN: 0001-0782. DOI: 10.1145/3446776.

[67] D. Zhou, M. Ye, C. Chen, T. Meng, M. Tan, X. Song, Q. V. Le, Q. Liu, and D. Schuurmans, "Go wide, then narrow: Efficient training of deep thin networks," in *ICML*, 2020. arXiv: 2007.00811 [cs.LG].

**UNIVERSITÉ CATHOLIQUE DE LOUVAIN**
**École polytechnique de Louvain**
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**