5-2019

# Compressing Deep Neural Networks via Knowledge Distillation

Ankit Kulshrestha
*Clemson University*, ankitkulshrestha0912@gmail.com

# Compressing Deep Neural Networks via Knowledge Distillation

---

A Dissertation
Presented to
the Graduate School of
Clemson University

---

In Partial Fulfillment
of the Requirements for the Degree
Master of Science
Computer Engineering

---

by
Ankit Kulshrestha
May 2019

---

Accepted by:
Dr. Melissa C Smith, Committee Chair
Dr. Brian Dean
Dr. Adam Hoover

I am deeply grateful to my parents Dr. Sandeep Kulshrestha and Mrs. Neenu Kulshrestha who supported me throughout this work and whose encouragement spurred me to work harder each day. I thank my advisor Dr. Melissa Smith for her constant support, feedback and for shaping my thought and research ability. I also thank Dr. Brian Dean for taking the time out of his busy schedule and listening to my approaches while always encouraging me to refine my work to a high standard.

Finally, I thank my friend and colleague Eddie Weill, who has been a great friend, teacher and guide along the path of my graduate career at Clemson University.

# Table of Contents

# List of Tables

# List of Figures

# Abstract

There has been a continuous evolution in deep neural network architectures since Alex Krizhevsky proposed AlexNet [22] in 2012. Part of this has been due to increased complexity of the data and easier availability of datasets and part of it has been due to increased complexity of applications. These two factors form a self sustaining cycle and thereby have pushed the boundaries of deep learning to new domains in recent years.

Many datasets have been proposed for different tasks. In computer vision, notable datasets like ImageNet [7], CIFAR-10, 100 [21], MS-COCO [24] provide large training data, with different tasks like classification, segmentation and object localization. Interdisciplinary datasets like the Visual Genome Dataset [20] connect computer vision to tasks like natural language processing. All of these have fuelled the advent of architectures like AlexNet [22], VGG-Net [34], ResNet [12] to achieve better predictive performance on these datasets. In object detection, networks like YOLO [31], SSD [25], Faster-RCNN [32] have made great strides in achieving state of the art performance.

However, amidst the growth of the neural networks one aspect that has been neglected is the problem of deploying them on devices which can support the computational and memory requirements of Deep Neural Networks (DNNs). Modern technology is only as good as the number of platforms it can support. Many applications like face detection, person classification and pedestrian detection require real time execution, with devices mounted on cameras. These devices are low powered and do not have the computational resources to run the data through a DNN and get instantaneous results. A natural solution to this problem is to make the DNN size smaller through compression. However, unlike file compression, DNN compression has a goal of not *significantly* impacting the overall accuracy of the network.

In this thesis we consider the problem of model compression and present our end-to-end training algorithm for training a smaller model under the influence of a collection of "expert" models. The smaller model can be then deployed on resource constrained hardware independently from the expert models. We call this approach a form of compression since by deploying a smaller model we save the memory which would have been consumed by one or more expert models. We additionally introduce memory efficient architectures by building off from key ideas in literature [14, 4, 15] that occupy very small memory and show the results of training them using our approach.

# Chapter 1

# Introduction

Deep Neural networks have revolutionized the way we interact with data. They are currently being deployed in computer vision system, medical imaging systems and even speech recognition applications. As the abundance of datasets has grown, deeper neural networks have been proposed to achieve competitive performance on those datasets. However, the increase in predictive accuracy comes with the growing cost of model size. For example, ResNet-18 [12] has a size of **128 MiB** while a WideResNet [36], with same number of layers can go upto **442MiB**. Additionally, deeper neural network models take longer to process data once they have been trained (this is called inference time from here on). In real world deployment scenarios, it is critical that a trained neural network model has a low latency while processing real time input. Considering that modern deep neural networks are both large and have high latency, it is imperative that techniques to compress them are applied, which has given rise to the field of model compression. In this work, we will define the problem and explore techniques that have been proposed so far. We will then demonstrate our method and discuss the relative benefits and the trade-offs that we make.

The weights in any given neural network layer have high redundancy, i.e. there are several weights in the same layer which can be set to zero without any significant decrease in accuracy. Hence, many previous techniques have focused on removing the redundant weights to reduce the model size. Some of these techniques involve pruning, quantization or SVD decomposition. However, a drawback of removing redundant weights is that it sparsifies the neural network weight matrices, prompting the use of specialized storage formats like compressed sparse column (CSC) format. Depending on the

degree of sparsity, this could become a major limiting factor in the inference times of a compressed model. Recently, a newer paradigm of compression by knowledge transfer has been proposed. It seeks to avoid the drawback of sparsifying weight matrices and "compress" the knowledge in a larger model into a smaller model such that the smaller model has comparable accuracy to that of a bigger model. We will discuss these techniques in detail in Chapter 3.

## 1.1 Contributions

In this work, we chose the newer paradigm of compressing neural network models by knowledge transfer by considering the formulation of Distillation Loss proposed by Hinton, *et al.*[13]. The original implementation assumes a single teacher student paradigm of transferring knowledge by proposing a loss function that encourages the student to produce a similar softmax distribution over the classes as a teacher model. This transfer of "dark knowledge" is achieved by "softening" the softmax probability distribution by a constant temperature $T$.

We extend this loss function to incorporate the knowledge from multiple teachers into a single smaller student model. Additionally, we do not assume the temperature $T$ to be a uniform constant and show analytically and experimentally that it outperforms the performance of a student model trained with Distillation Loss. More importantly, our proposed loss function shows that it is possible to control the amount of distillation of knowledge from each teacher.

We introduce two new student model architectures in this work as well. These architectures are oriented to be as memory efficient as possible, while still having comparable accuracy. The focus is on mobile deploy-ability and the ability to learn from multiple teachers. We describe them in detail in Chapter 4.

In order to evaluate the results we perform experiments with both student architectures on the CIFAR-10 [21] dataset. To further validate our hypothesis, we perform similar experiment with smaller student models on the SVHN [28] dataset. In other related approaches to knowledge distillation, there has been a lack of focus on the mechanism of knowledge transfer between two different networks. We experimentally examine this question by modifying a standard dataset and removing

instances of a particular class from the student training set. We then compare the rate of misclassification achieved by the approach proposed by Hinton *et al.*and our approach. These results are presented in Chapter 5.

## 1.2 Thesis Overview

This work is organized into roughly two parts. The first part comprising of Chapters 2-3 comprehensively introduces the background on model compression and surveys some of the existing approaches in the field. The second part comprising of Chapters 4-5 provides an in-depth view of our contributions to model compression.

Chapter 2 introduces the supervised learning paradigm in detail. It details the perceptron model of computation and then builds off of it to introduce the concept of non-linear activation functions. Multilayer perceptrons are discussed in the next section, followed by a general overview of the stochastic gradient descent algorithm. Chapter 3 motivates and formalizes the problem of model compression before examining some of the major approaches like Optimal Brain Damage [6], Weight Approximation via SVD Decomposition [8], etc. We present our own approach along with proposed student architectures in Chapter 4. Experimental simulations and results are discussed in Chapter 5. Finally, we conclude our work and offer suggestions for future work in Chapter 6.

# Chapter 2

# Background

This chapter introduces the background material for understanding the supervised learning paradigm and the need for model compression. Section 2.1 introduces the overall supervised learning paradigm. Section 2.2 introduces the McCulloch Perceptron model and Section 2.3 builds on that to introduce the concept of non-linearities and activation functions. Finally, Section 2.4 introduces the Multi Layer Perceptron and Section 2.5 formally defines the stochastic gradient descent algorithm.

## 2.1 Supervised Learning

Supervised learning is a machine learning paradigm in which the model under training is explicitly provided with correct labels for each data point. Formally, in the supervised learning paradigm in which sample pairs $(x, z)$ are drawn from a training set $\zeta$ during the training phase of the algorithm. Each sample pair consists of a data point and an associated correct label. Additionally, a disjoint set of samples is drawn from the same input data distribution and is not presented to the model during training. Instead this is used to get a measure of how accurately the model maps the input data to the given correct labels. This set is called *validation set* and this ability of a machine learning model to learn from a small distribution and make predictions on unseen samples drawn from a similar distribution is called *generalization*. The training task generally attempts to minimize an *error function* $E$ which is defined as the distance between the predicted value $y$ and $z$. The error function is sometimes referred to as a loss function. We shall describe some common loss functions when we discuss Neural networks in Section 2.4

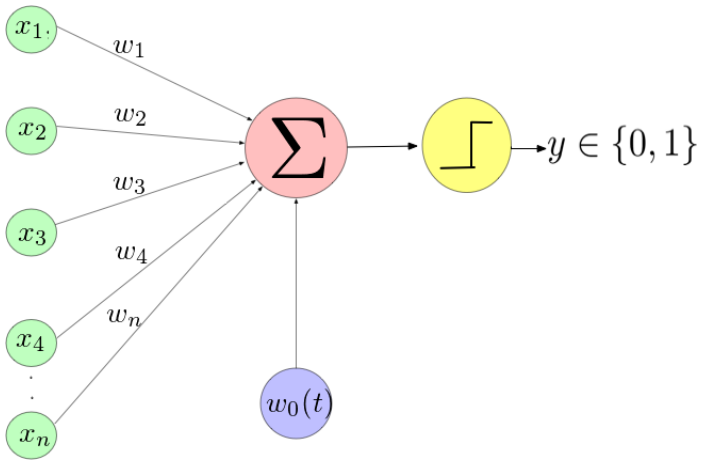


Figure 2.1: The Perceptron

There are two broad subcategories of tasks in the supervised learning paradigm - linear and logistic regression. The two tasks differ in the way they train the model to map the inputs to the outputs. Logistic regression aims to map a set of inputs $x \in \mathbb{R}^n$ to a set of distinct classes $c \in \mathbb{C}$. This can be thought of a bin-assignment problem in which ranges of continuous inputs are mapped to discrete bins. Linear regression on the other hand, attempts to map a set of inputs $x \in \mathbb{R}^n$ to a continuous output variable $\Theta$.

## 2.2 The Perceptron Model

We have reviewed the theoretical framework of traditional approaches to machine learning in the previous section. Before we discuss the theoretical framework of neural network learning and optimization, it is imperative to discuss the perceptron algorithm.

Figure 2.1 shows the perceptron computation diagram. We define the input vector $X = \{x_1, x_2....x_n\}$ to be an ordered set of the dimensions of the data which may or may not be conditionally independent of each other. The weight vector $W = \{w_1, w_2....w_n\}$ determines the "degree of importance" of each individual input dimension. We denote the output of the cell marked $\Sigma$ to be $y$. With this information we define:

$$y = w_1 * x_1 + w_2 * x_2 + ...w_n * x_n + w_0(t) \tag{2.1}$$

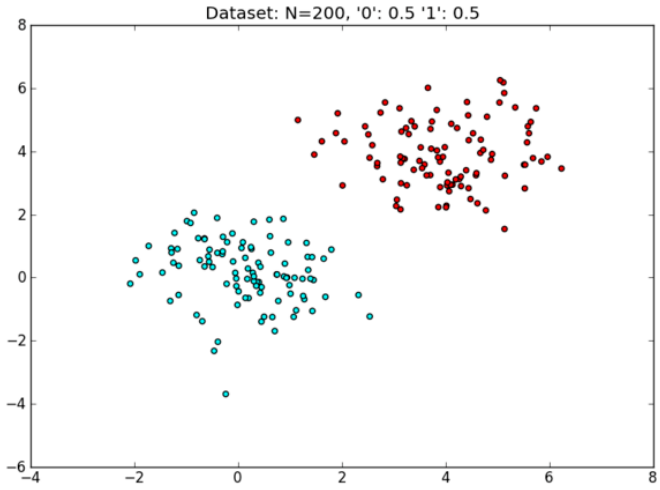


Figure 2.2: An example of linearly separable data in 2 dimensions

Here, $w_0(t)$ is called the bias. Equation 2.1 can be more succinctly written as:

$$y = \sum_{i=0}^{N} W * X^T + w_0(t) \tag{2.2}$$

Equation 2.2 is similar to the equation of a straight line in two dimensions. Effectively, it gives a "hyperplane" which divides the input space into two linearly separable halves. The perceptron algorithm does not assume the weight vector $W$ to be fixed. Instead, the weights are *tunable* parameters that get trained using a training set. Like most superivsed learning algorithm, the training set for a perceptron comprises of $(x_i, z_i)$ pairs where $z_i \in \{0, 1\}$. For each data, label pair we have a set $Y = \{y_1, y_2, y_3.....y_n\}$ where each $y_i$ is computed via equation 2.2. The training proceeds under a certain learning rule, which relies on computing an *error measure* between the computed samples $y_i$ and label $z_i$. If the predicted output matches the correct label, then no action is performed, else the weight vector $W$ is adjusted as:

$$\hat{w_i} = w_i - \eta * (z_i - y_i) * x_i \tag{2.3}$$

Equation 2.3 is the Delta learning rule. The training continues by making several passes on the training set and continuously testing on the test set. The algorithm is said to have converged when the error in classification is below the tolerance threshold. It must be noted here that the perceptron model assumes that the data is linearly separable and only consists of two classes. To maintain rigor,

we define the data being linearly separable when the following is true for all data points:

$$y = \begin{array}{ll} w * X^T + w_0(t) > 0 & x_i \in X_0 \\ w * X^T + w_0(t) < 0 & x_i \in X_1 \end{array} \tag{2.4}$$

In Equation 2.4, the set $X_0$ corresponds to all $x_i \in X$ whose label is 0 and $X_1$ corresponds to all $x_i$ whose label is 1. For example, in figure 2.2 the two data points cluster in two different positions. We can assign label 0 to all green points and label 1 to all red points since they fall on either side of a line $y = -x$. The perceptron training algorithm seeks to *minimize* the error function $E$ to the least value possible over the input space. However, the perceptron framework is too rigid and only allows for binary classification over linearly separable data.

## 2.3 Towards Neural Networks

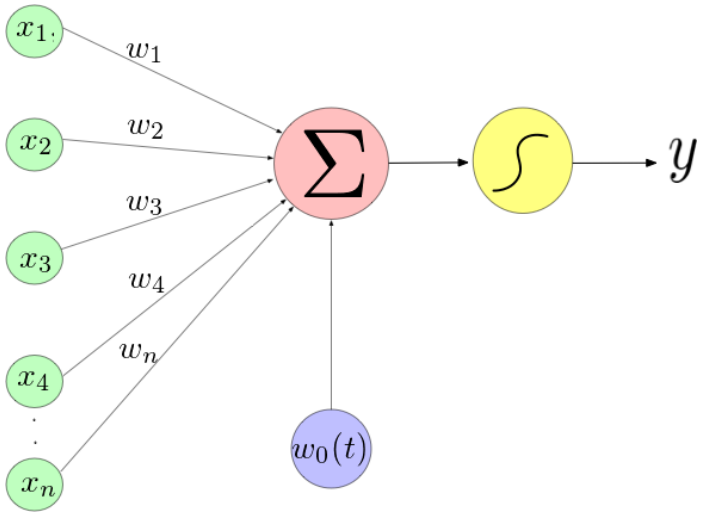


Figure 2.3: The Neuron Model

To overcome the limitations of the perceptron model, we now explore the neuron model of computation and generalize it to the case where the neurons are not individual computation unit, but are abstracted into a bigger computation unit called *layer*. Finally, we explore multilayer perceptron model.

The neuron computation model is shown in Figure 2.3. It is very similar to the perceptron model

in terms of the input vectors $X$ and the weight vectors $W$. However, it differs from the perceptron model in one very significant aspect. In the perceptron, the output of the model was a binary class value. In the case of neuron however, $y = 1$ if and only if the value from the cell before it is greater than a threshold. The cell before $y$ is called an *activation function* or simply activation and is responsible for turning the real valued input of the $\Sigma$ cell into a non-linear function of $X$ between a certain fixed range. We will discuss the activation function in more detail in Section 2.4.1. Denoting the activation function as $\sigma$, we compute $y$ as:

$$y = \sigma(\sum_{i=0}^{n} W * X^T + w_0(t)) \tag{2.5}$$

If $\sigma(x)$ is greater than some threshold $t$ then the neuron is said to be in the "firing" state. The resulting $y$ is then a real value bounded by the upper and lower bounds of the activation function and is fed forward to other directly connected neurons. We will introduce another computational abstraction in our simplistic model when we discuss multilayer perceptrons in Section 2.5.

### 2.3.1 Activation Functions

We have discussed the significance of activation function in the neuron computation model in the previous section. In this section, we define and describe many commonly used activation functions in modern deep convolutional neural networks. However, before discussing the individual activation functions, we define the desirable properties that are required for a mathematical function to be used as an activation function:

1. It should have a strictly defined upper and lower bound i.e. any output value produced by this function can approach the limits but never *attain* the limit and beyond.

2. It should be continuous and differentiable for all input $x$.

We discuss some commonly used activation functions in modern CNNs in the next subsections

#### 2.3.1.1 The Sigmoid Nonlinearity

The sigmoid nonlinearity is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

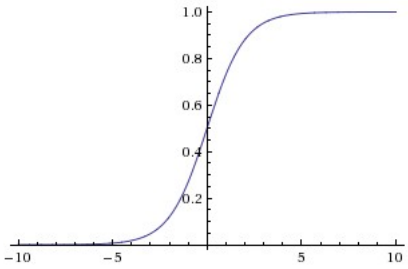


Figure 2.4: The sigmoid nonlinearity

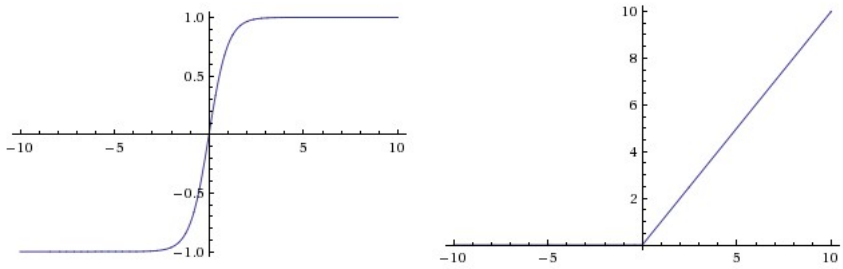


Figure 2.5: The Tanh and RELU nonlinearities

The gradient $\frac{\partial\sigma(x)}{\partial x}$ is simply $\sigma(x) * (1 - \sigma(x))$. Figure 2.4 shows the graphical representation of the sigmoid function. It can be seen that the output is constrained to be between $[0, 1]$. However, this function is not widely used in modern deep CNNs since it has a tendency to *saturate* and kill the gradients, i.e. when the sigmoid function saturates to produced only 1 as a response to any input, then the gradients approach zero. In deep neural networks this can be an issue since the weights in the lower levels of the network don't get updated. Another key factor is that the response is not *zero centered* which can cause the sign of the gradient to change in a zigzag manner during backpropagation.

#### 2.3.1.2 The Tanh and ReLU nonLinearities

Figure 2.5 shows the graphical representation of the Tanh and ReLU nonlinearities. The Tanh nonlinearity is defined as:

$$tanh(x) = \frac{1 - e^{-x}}{1 + e^{x}} \tag{2.7}$$

This function can be simplified to $tanh(x) = 2 * \sigma(x) - 1$. This formulation makes it easy to calculate the gradient using standard differentiation and the definition of gradient of $\sigma(x)$. A key point of difference in $tanh(x)$ and $\sigma(x)$ is that the former constrains the output values to be between $[-1, 1]$

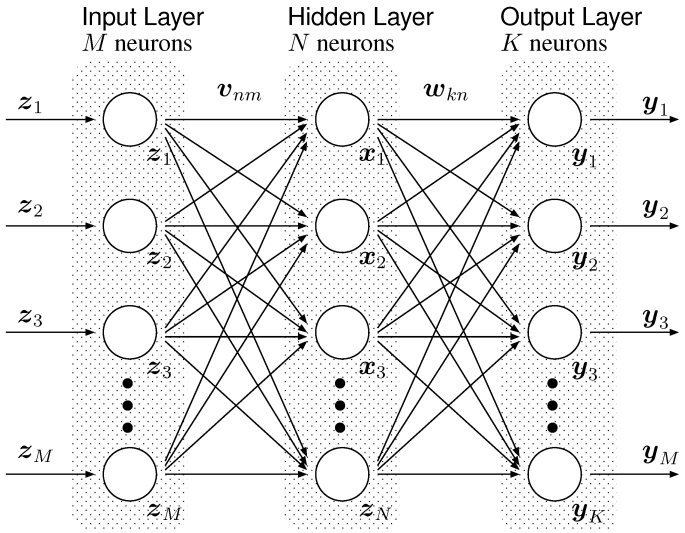


Figure 2.6: Multilayer Perceptron model

and is zero centered. Hence this nonlinearity is almost always preferred in place of the latter.

The ReLU nonlinearity is defined as $relu(x) = max(0, x)$. Thus any $x < 0$ is simply quashed to 0. This nonlinearity was shown to be extremely helpful in allowing a neural network to converge via gradient descent (we will discuss this shortly) by Krizhevsky, *et al.* in [22]. Computationally, the ReLU units are a simple thersholding operation that is very simple to compute during different phases of the training. However, the units are susceptible to producing an output of 0 if proper care is not taken during training.

## 2.4 Multilayer Perceptrons

In the previous sections we have seen the computational model for a perceptron and neuron. In this section we introduce a computational abstraction over a neuron and then introduce the multilayer perceptron. We then describe in detail the two training passes in a multilayer perceptron. However, before we start it must be noted that the term multilayer perceptrons is a bit of a misnomer since the individual units are neurons.

We have seen the computation of the output $y$ in a neuron. While this is feasible for one neuron and one dimension, it quickly becomes computationally infeasible for data with thousands of dimensions

and corresponding neurons. We thus introduce the computational abstraction of a *layer* which is defined to be a collection of neurons, such that if one computes the output of a layer, they would have computed the output of each individual neuron. We represent a layer by a matrix of $(m+1) \times 1$ dimensions where $m$ is the number of neurons in that layer. The additional neuron is the constant "bias" that we saw earlier.

Multilayer perceptrons are typically organized into layers with each neuron in Layer $L_i$ connected to each neuron in Layer $L_j$ for $j > i$. If we consider this model to be a black box then only the input layer and the output layer are visible to the user. The inner layer through which the feature vector passes are hidden. Hence these layers are called *hidden layers*. A neural network (henceforth used in place of multilayer perceptron) follows the composability principle, i.e. it can be composed of various layers stacked one after the other (in the horizontal point of view). It has been observed that there is a direct correlation between the performance of the neural network and the number of hidden layers in between the input and output layers. In Figure 2.6 we show a neural network comprising of one hidden layer.

#### 2.4.0.1 Forward Pass

Neural networks are parametrized algorithms, i.e. they try to estimate the maximum log likelihood $P(y|x;\theta)$ such that it matches the expected values. The parameter $\theta$ can be interpreted as the weights and the biases of each layer in a neural network. From a high-level perspective, a neural network then attempts to *approximate* a universal function defined over the input space by tuning these parameters. An interesting property of neural networks is that they can be described in terms of both graphs and equations. We show a graphical view of a simple neural network in Figure 2.6. Equation 2.10 shows the forward pass in a multi-layer perceptron. An implicit assumption in the equation is the use of the sigmoid nonlinearity $\sigma(x)$. We refer to the output of a layer as $u_{ij}$ indicating that it's the *ith layer connected to the jth layer*.

Computing the forward pass in a neural network is essentially a vector operation where we compute the matrix-vector products of the input vector (to a layer) and it's parameters. We have previously defined the input to be an $(m+1)$ dimensional vector. A layer also maintains it's own parameters $\theta^l$ which are it's weights and biases. Since each neuron in layer $i$ connects to every neuron in layer $j$ we

can define the weights $w_{jk}^{l}$ to be the weights of layer $l$ connecting weight $k$ of layer $l-1$ to weight $j$. More succinctly, the layer weights can be represented as a weight matrix of dimensions $\mathbb{R}^{M\times N}$where $M$ refers to the number of neurons in layer $l-1$ and $N$ refers to the number of neurons in layer $l$. For the network shown in Figure 2.6 we compute the forward pass according to the equations below (we don't assume a constant bias node):

$$Y_1 = X * W_1^T \tag{2.8}$$

$$Y_2 = Y_1 * W_2^T \tag{2.9}$$

$$\hat{Y_o} = Y_2 \tag{2.10}$$

$\hat{Y_o}$ is the output vector produced by the neural network. The number of weight matrices are equal to the number of layers in the neural network with the exception of the input layer. Thus for our 3 layer network we have 2 weight matrices. Now since there were $m$ inputs in the input layer and it was connected to a layer with $N$ neurons, $W_1$ has dimensions $\mathbb{R}^{N\times M}$. Similarly, in the output layer there were $k$ neurons hence $W_2$ has dimensions $\mathbb{R}^{K\times N}$. It is also evident from the equations that the forward pass is a staged matrix vector product that ultimately produces parameters for either classifcation or regression. If the network in figure 2.6 was being used for classification, then we would define a function *softmax* as:

$$\zeta = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.11}$$

Equation 2.11 normalizes the output vector $\hat{Y_o}$ into probability values between [0, 1]. We could then predict the most likely class as $\hat{y} = argmax(\hat{Y})$.

## 2.5 Stochastic Gradient Descent

In the previous section we've discussed the computational model of a multilayer perceptron. We presented a way to compute the output of the neural network by succesively applying matrix-vector products to each layer. However, this alone does not solve the problem of finding the minima for some objective function $J(\theta)$. In this section, we introduce Gradient Descent and present a brief overview of how modern deep neural networks are trained via this algorithm.

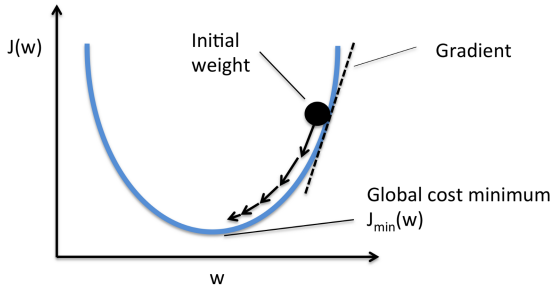


Figure 2.7: A two dimensional view of gradient descent

### 2.5.1 Gradient Descent

Figure 2.7 graphically represents the process of gradient descent in two dimensions. The ball at the top of parabola aims to ultimately travel to the bottom of the curve, which also coincides with the minima. Formally, let the cost function $J(\theta)$ define a non-convex optimization surface (i.e. it can have multiple local minima and maxima) over $N$ dimensions and let $\theta$ completely determine the value of $J(\theta)$, then we define gradient descent as an iterative algorithm, that minimizes the cost function by updating values of $\theta$ at each step. We say that the algorithm has converged when for any $\theta - \delta$: $J(\theta - \delta) = J(\theta)$. At each step $t$ the parameters are updated according to the equation:

$$\hat{\theta} = \theta - \eta * \frac{\partial J(\theta)}{\partial \theta} \tag{2.12}$$

In Equation 2.12, $\eta$ is known as the *learning rate*. In Figure 2.7 this corresponds to the initial push that is provided to the ball so that it rolls down the curve. In gradient descent this is one of many different constant that are set *before* a network is set to training. In practice, care must be taken to select $\eta$ correctly since a high learning rate will cause the network to overshoot the minima while a too low learning rate will cause the opposite effect.

In Equation 2.12 we note that the next value of a parameter $\theta$ also depends on it's rate of change w.r.t the cost function. For the algorithm to converge, *all* $\theta$ is every layer must be updated. An efficient way to compute this is introduced by the "Backpropagation Algorithm" which is an algorithm to compute the update for each layer's parameters by successively applying the chain rule of differentation.

**Algorithm 1** The Stochastic Gradient Descent Algorithm

1. Select input training data $\mathcal{D}_n$ containing data label pairs. $x \in \mathbb{R}^{hxwxc}$ and $y \in \{1...m\}$ for $m$ possible classes.

2. Select a network $\Gamma$ to be trained with an objective function $O$.

3. Select a network optimizer $\Omega$.

4. Compute the forward pass for $i = 0 \rightarrow n$ or convergence.

**procedure** FORWARD PASS($\mathcal{D}_n, \Gamma, O$)
  **while** $(x, y) \in \mathcal{D}_n$ **do** ▷ Compute the forward loss function given label and predictions
    $\hat{y} \leftarrow \Gamma(x)$
    $loss \leftarrow O(\hat{y}, y)$
  **end while**
**end procedure**

5. For each iteration $i \neq n$ compute the backward pass.

**procedure** BACKWARD PASS($loss, \Gamma$)
  $loss^{'} \leftarrow \frac{\partial loss}{\partial O}$
  **for** $l = n - 1 \rightarrow 0$ **do** ▷ Compute gradients for all layers in reverse
    $l_\alpha \leftarrow \frac{\partial loss^{'}}{\partial \alpha}$
    $\Gamma[l] \leftarrow \frac{\partial l_\alpha}{\partial x}$
  **end for**
**end procedure**

6. Update the weights across the neural network using Equation 2.12 and optimizer $\Omega$

Algorithm 1 provides an overview of the stochastic gradient descent (SGD) algorithm for a classification task. The algorithm proceeds until the training iteration reaches the maximum iteration or we find that the training has converged. A training procedure in SGD is said to have converged when at an iteration $i + 1$, $\theta_{i+1} = \theta i$. At each training step, the algorithm requests a batch of data and computes the most likely output for that batch. Prior to any training procedure, we define a `Loss Function` that measures the dissimilarity between the predicted and the actual ground truth values. In a training step, this loss function is used to compute the *error*. In the backward pass we compute the gradient of the loss. Then for each layer $l$ in our network $\Gamma$ we compute the gradient of the loss w.r.t the *activation function* and then compute the gradient of this value wrt the input. Since this computation proceeds from the outermost layer to the input layer, this is also referred to "reverse mode differentiation" in the literature.

We have mentioned that at each training step, the algorithm requests *batches* of data. The reason for this is that modern datasets contain data that cannot be fit on a single computational unit. Hence, we divide the data into pre-determined size called *batch_size*. Varying this parameter, has a direct effect on the time it takes the SGD procedure to converge (assuming the data has a local minima). The larger the batch size the faster the convergence. Additionally, the batches are shuffled and chosen in a *stochastic* manner, i.e at any given training step a random batch of input patterns can be considered. This prevents a phenomemon called *overfitting* in which a neural network achieves high accuracy on the training set while having a poor accuracy on the unseen testing set.

## 2.6 Chapter Summary

In this chapter we defined the supervised learning task and formally defined two types of tasks that fall under this family of learning approaches. We then looked at different types of learning units that have been proposed to learn the mapping between a given input pattern $X$ and desired output pattern $Y$, starting from a simple perceptron to more complex architecure of a multi-layer perceptron. Finally, we looked at an algorithm to stochastically estimate a minima starting from a random initialization point and computing forward and backward variables across layers in a network. In the next chapter, we examine the problem of model compression and look at different approaches that have been proposed so far to solve it.

# Chapter 3

# Introduction to Model Compression

In this chapter we discuss the motivation for model compression. We then formalize the problem and then present some existing approaches for model compression. We present our own findings in chapter 4 and illustrate some of the extensions to our work in Chapter 5.

## 3.1 Motivation

Deep learning has seen exponential growth in the last five years. Part of it has been due to the rise in the computational capability of Graphics Processing Units (GPUs) which had been erstwhile seen as essential to video games and building graphical simulations of real world models. Additionally, the growing amounts of data in different domains have also fuelled the development of more and more complex deep neural networks(DNNs) that can accurately generalize to unseen data in those domains.

Apart from scientific research, DNNs have found applications in consumer applications as well. From face recognition in selfies to digital voice assistants like Siri, there is a growing demand for instant access to highly accurate models. The ideal approach would be to deploy the DNN on the device and reduce any communication latency from a data server. However in practice, this means that

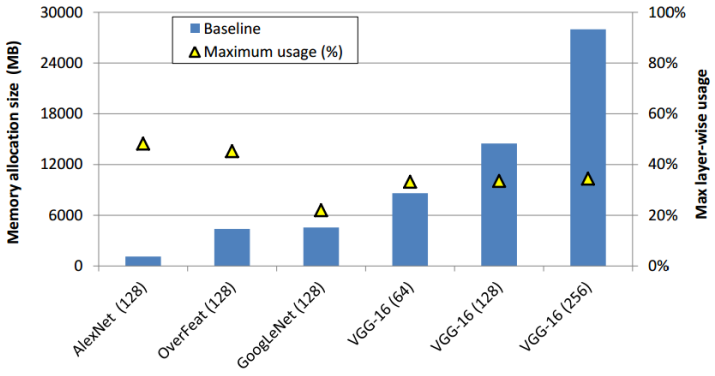


Figure 3.1: Memory consumption of some popular DNN architectures.

we need to somehow have the model in the DRAM of the device itself. This is difficult to achieve out of the box since as the complexity of data and the associated model grows, it directly impacts the size and the energy consumption of the model. This motivates the need for having some way of decreasing the memory footprint of a trained model *without* affecting the overall accuracy of the network. Figure 3.1 shows the memory consumption of some successful deep convolutional neural networks on the Imagenet [7] dataset. It can be observed that the high accuracy of a DNN is highly correlated with a higher memory consumption. In a neural network the main sources of memory consumption are the tunable parameters (weights, biases) of a given model. The deeper the model, the more number of tunable parameters are needed to optimize the nework on a given dataset.

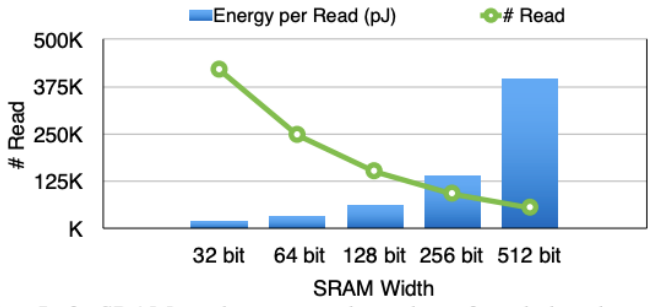


Figure 3.2: Energy Consumption per Memory access

Energy cost of common read operations in different precisions in a neural network is shown in Figure 3.2 [9] [10]. Most neural networks are trained in full precision i.e. the word size is 128 bits or more. From the figure, it can be seen that the energy per read operation significantly increases as

the word size is increased. Thus, from an energy point of view, having a full precision network on a resource constrained device could lead to faster battery drain and decreased performance.

As a result, we formalize the model compression problem as follows: Given a neural network model trained on a dataset $\mathcal{D}$ consisting of $N$ training samples $(X_i, Y_i)$ with $\Gamma$ parameters and having an accuracy $\alpha$ on the test set drawn from the same dataset, obtain a neural network model with $\hat{\Gamma}$ parameters and accuracy $\hat{\alpha}$ such that $\hat{\Gamma} << \Gamma$ and $\hat{\alpha} \approx \alpha$.

We discuss approaches to this problem through the rest of this chapter.

## 3.2 Saliency Based Network Pruning

The earliest approaches to model compression were developed keeping the multilayer perceptron architecture in mind. Implicitly, that assumes that each neuron in layer $j$ is connected with each neuron in layer $i$ and weighted as $w_{ji}$. Additionally, it also assumes that the network was sufficiently shallow to allow a backward pass to be computed relatively easily. Abstractly, the approaches are focused on somehow computing a measure of *redundant* weights in neural networks. From our discussion of the SGD algorithm, we know that a network has converged when $\theta_l + \epsilon \approx \theta_l$. It then follows, that during the course of weight updates there may be some parameters in the same layer that converge to same value, no matter their initial point. Hence, removing one redundant parameter should not (in theory) affect the accuracy of the neural network. Moreover, one can compensate for the loss of a redundant weight by increasing all others by a proportional amount. However, computing *which* parameter to remove is a hard task. In this section we examine some approaches that have been proposed to compute not only the redundant weights but also a measure of their *saliency*.

### 3.2.1 Hessian Based Approaches

Some of the earliest approaches to network pruning involved perturbing the trained parameter by a small amount $\Delta\theta$ and then measuring the change in the overall objective function $\delta E$. For simplicity in analysis we assume that the only trainable parameters in the network are the weights $w$ and we are perturbing them by $\Delta w$. The change in the objective function is obtained by a *Taylor Series*

*Expansion* and ignoring the higher order terms as:

$$\delta E = \sum_i g_i \delta w_i + \frac{1}{2} \sum_i h_{ii} \delta w_i^2 + \frac{1}{2} \sum_{i \neq j} h_{ij} \delta w_i \delta w_j + O(||\delta w||^3) \tag{3.1}$$

In Equation 3.1 $g_i$ are the element of a vector $\mathbf{G} = \frac{\partial E}{\partial w}$. The second and third terms comprise of $h_{ii}$ and $h_{ij}$, which are elements of a **Hessian** matrix $\mathbf{H} = \frac{\partial E^2}{\partial w_i \partial w_j}$. The information contained in the *inverse* of Hessian matrix elements can help determine the saliency of the weights to be pruned. We explore two main works in the following sub-sections:

#### 3.2.1.1 Optimal Brain Damage

In Optimal Brain Damage [6], LeCun,*et al*. propose a method to compute the terms in equation 3.1 in an efficient manner. It assumes that the total change in the objective function must be caused only by the changes in the parameters themselves, i.e. $\delta E$ is solely the sum of pertubations in $w_{ii}$. Thus, any cross terms are neglected. Based on this assumption, Equation 3.1 reduces to:

$$\delta E = \frac{1}{2} \sum_i h_{ii} \delta w_{ii}^2 \tag{3.2}$$

Additionally, the perceptron model is considered to be *shared weights*, i.e. $w_{ij}$ is same for any connection from neuron $i$ to $j$. If $V_k$ is the set of connectivity indexes for a neuron $k$, then the diagonal elements $h_{ii}$ can be computed as:

$$h_{ii} = \sum_{(i,j) \in V_k} \frac{\partial E^2}{\partial^2 w_{ij}} \tag{3.3}$$

The overall procedure for Optimal Brain Damage is Algorithm 2:

**Algorithm 2** Optimal Brain Damage Procedure

1. Train the network using gradient descent until convergence.
2. Compute $h_{ii}$ using Equation 3.3
3. Compute saliencies $s_{ii} = h_{ii} w_{ij}^2 / 2$
4. *saliency_list* $\leftarrow$ SortSaliency(saliencies)
5. Eliminate weights with low saliencies in the *saliency_list*.
6. Iterate to step 2 until there is no more change in in the hessian.

The main computational bottleneck stems from steps 3 and 4 in the procedure. During every backward pass, the procedure requires knowledge of *every* weight connection from neuron $i$ to $j$ and

then performs an explicit sorting step, before pruning the low saliency weights. Another similar idea was proposed Karnin *et al*.in [18] where instead of computing the Hessian, a shadow saliency list of weights is built and then low saliency weights are removed. In the context of modern deep neural networks, this bottleneck becomes a significant source of affecting both training and inference times.

#### 3.2.1.2 Optimal Brain Surgeon

In [11], Hassibi *et al*.propose a more general procedure called Optimal Brain Surgeon. Unlike OBD, this method does not assume that the weight change arises due to self connections only. They build from Equation 3.1 and show that the general solution for computing $\delta w$ given the constraint of deleting one weight is:

$$\delta w = -\frac{w_q}{H_{qq}^{-1}}.e_q \tag{3.4}$$

$$L_q = \frac{1}{2}\frac{w_q^2}{H_{qq}^{-1}} \tag{3.5}$$

Here, $e_q$ is a unit vector in the $qth$ direction in weight space and $H^{-1}$ is the *inverse* of the Hessian matrix. $L_q$ is the saliency measure for the $qth$ weight. The optimal brain surgeon procedure is described in Algorithm 3.

**Algorithm 3** Optimal Brain Surgeon Procedure

1. Train a multilayer perceptron for a given data until convergence.
2. Let $E$ be the learned energy, compute $H^{-1}$.
3. Find $q$ that belongs to $min(L_q)$. (Equation 3.5)
4. Delete $w_q$ and update other weights by $\delta w$. (Equation 3.4)
5. Repeat Step 2.
6. Stop when $\delta w_t \approx \delta w_{t+1}$, where $\delta w_t$ is the weight change at time $t$.

The most significant (and time consuming) step in this procedure is step 2. A recursive method to compute step 2 has been proposed in [11]. Since the procedure is recursive, the memory requirement for computing the inverse Hessian will considerably increase as the network depth is increased. Another assumption made by the Hassibi *et al*. is that there is *total connectivity* in the network,i.e. all neurons in layer $i$ are assumed to be connected to all neurons in layer $i+1$. However, this is not the case in convolutional layers where there is a *spatial* connectivity in a fixed region. Hence,

if a low saliency weight is deleted in one region, it will be difficult to update the weights in other regions.

## 3.3 Deep Neural Network Compression

In this section, we discuss two techniques that have extended the goals of the previous approaches to modern deep neural network. Most of these techniques have been proposed for convolutional neural networks, but the same principles apply to other kinds of networks as well. A common theme that unifies both the approaches is that they both exploit some inherent property of a given network to remove redundancy in a computationally efficient manner.

### 3.3.1 Deep Compression

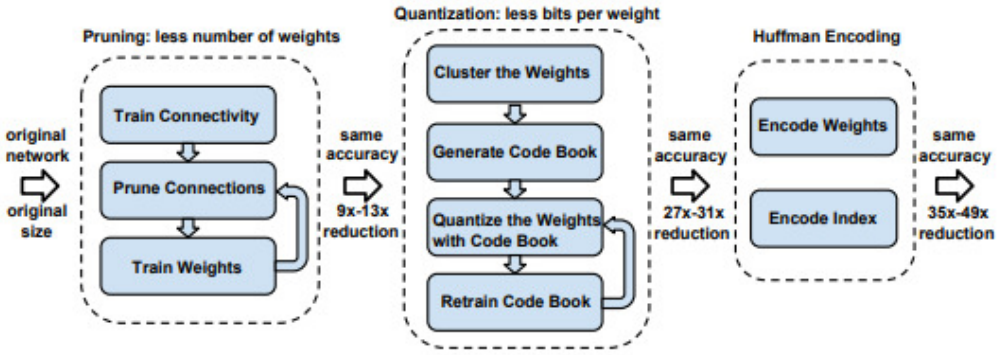


Figure 3.3: The Deep Compression Pipeline

Figure 3.3 shows the compression pipeline proposed by Han, *et al.* in [9]. As can be observed from the figure, there are three distinct stages, sequentially arranged to compress a given DNN. The first stage in the pipeline involves *iterative pruning*. We have already discussed the necessity of pruning and how by removing redundant weights, it can lead to a smaller model size. In this procedure, the first step is to learn the connectivity, i.e. the magnitude of weights $w_{ij}$ from neuron $i$ to $j$. An arbitrary threshold is chosen and any weight that is below the threshold is then pruned.

Like previous approaches, we note that changing the connectivity by $\delta w$ needs to be compensated by the same amount in other weights. In this approach, this step is performed by *retraining* the network

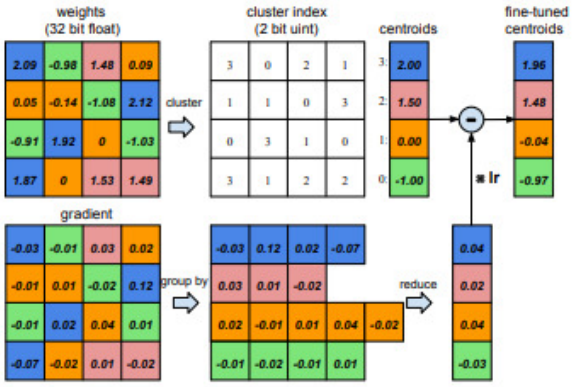


Figure 3.4: Trained Quantization in Deep Compression.

to learn the new connectivity. The second stage in Figure 3.3 involves *trained quantization*(Figure 3.4). During this stage, $n$ bins are chosen and $n$ centroids are initialized. Then, in a kmeans like procedure, weights are quantized as shown in Figure 3.4 to one of the $n$ bins. During the backward pass, the gradients are also grouped together, and the weight update is performed according to Equation 2.12. Finally, the centroids are updated as well. The resulting weights then have a direct mapping to a particular bin $i \in \{1...n\}$. Finally, the indices of the bins and the code book can be further compressed via Huffman Encoding.

#### 3.3.1.1 Drawbacks

The proposed approach is not as computationally intensive as building sensitivity lists and pruning. However, it does have its demerits. We note that the first stage involves setting an arbitrary threshold, and then pruning the connections. Since this threshold is arbitrary, in some cases it may lead to necessary weights being pruned as well. Another drawback is the explicit re-training step after pruning. In many cases, especially in production there may not be enough training time available to accommodate the re-training of the remaining weights.

While trained quantization is able to quantize continuous weights into discrete bins, it leads to *sparsity* in the network. While a sparse weight matrix occupies smaller space, it requires a special format (CSR, CSC) to be represented in memory. However, most high performance libraries like LaPack [2], BLAS [1], Intel MKL [16], cuBLAS [5] do not support indirect lookups for matrix-matrix and matrix-vector computation.

### 3.3.2 Weight Approximation by Tensor Decomposition

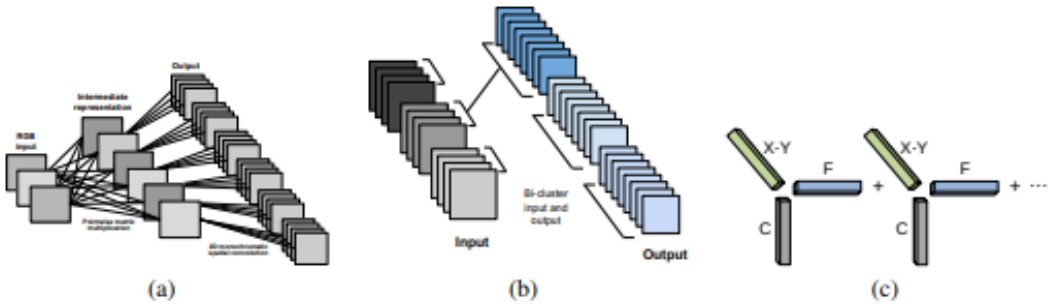


Figure 3.5: Approximation of Weight Tensors to Rank-1 Tensors

An approach to exploit linear structure in convolutional layers and approximate weight tensors is introduced by Denton,*et al.* in [8]. The central idea of the approach is to approximate the weight matrix of a $W$ with a close enough matrix $\hat{W}$. The tensor $\hat{W}$ then can be computed much more efficiently and have similar accuracy as before. The overall flow of the approach in shown in Figure 3.5.

The decomposition is first proposed for a low order (2 Dimensions) weight matrix $W \in \mathbb{R}^{mxk}$, which is approximated as $W = USV^T$. An assumption made in the paper is that the singular values for this type of decomposition decays rapidly and hence only the first $t$ entries can be considered signficant. For higher order tensors, i.e. $W \in \mathbb{R}^{mxnxk}$ the approach is adapted by folding the dimensions other than the first into one i.e. $W \in R^{mx(nk)}$. This can also be seen from Figure 3.5 where the higher order weight tensors are ultimately converted to rank-1 tensors.

It should be noted that this approach does not involve any re-training of the network at all. However, it exclusively focuses on removing the redundant weights in the convolutional layers while resting on the assumption that the singular values decay rapidly enough. Additionally, it does not discuss this approach in context of fully connected layers in a CNN, which acutally contribute to majority of the space requirement.

## 3.4 Compression By Knowledge Transfer

All approaches considered thus far have a common underlying theme - a trained network was the input to the algorithm and some sort of inherent property was exploited to prune, quantize, or approximate the weights. The output of the algorithms was the *same network* with significantly less weights. In this section, we discuss a different approach to compression, i.e. training a smaller dense network under the influence of an "expert" teacher. We describe some of the existing work in the domain of knowledge distillation and discuss their drawbacks. We discuss our own method and it's intuition in detail in the next chapter.

Before discussing related work in this area, we first define the notation of "teacher" and "student" model. A teacher model is a deep neural network (or an ensemble) that has a high performance on a given dataset. A student model, is another neural network, not necessarily as deep as the teacher and doesn't perform as well as the teacher on the same dataset. There are some benefits to this approach towards model compression. Firstly, there is no computational bottleneck of computing saliency values of connectivity between neurons $i$ and $j$ during training which leads to a faster training time. Secondly, there is no re-training involved. The student network occupies lesser memory, and once trained can be deployed anywhere. Thirdly, it doesn't make any assumptions on the underlying rank or eigenvalues of the weights in the student, since the student is train in a conventional manner by gradient descent.

### 3.4.1 Knowledge Distillation

In [3], Caruana,*et al*. train a smaller network by drawing synthetic samples from a data distribution that mimicked the one on which a larger expert model was trained. More formally, they propose an algorithm to approximate a joint probability distribution $p(\theta_s)$ that mimics the probability distribution of $\mathcal{D}_n$ where $\mathcal{D}_n$ is the training set of the expert model. The training samples for the smaller model are then drawn from this distribution with an a-priori probability of $p(x|\theta_s)$. The smaller model then is the "compressed" version of a larger teacher model. However, this procedure depended heavily on the training data for the larger model, and hence if that were to change a retraining of the smaller model would be required.

In [13], Hinton, *et al*. showed that it is not necessary to approximate the data distribution. Rather, they proposed an end-to-end knowledge distillation framework with a loss function called *Distillation Loss*. In the framework, there is a large complex model (which is assumed to be pre-trained) and a smaller "student" model. The large complex teacher model's output is used as a *soft target* for the smaller model and the overall loss function is proposed in Equation 3.6.

$$L_d = \alpha KL(o_z, vi) + (1 - \alpha)CE(o_z, y_i) \tag{3.6}$$

Here $v_i$ is a *softened* softmax probability at a particular temperature $T$ and $o_z$ is the output logits produced by the student model. Finally, $y_i$ are the actual "hard labels" obtained from the labelled dataset. We shall discuss softening temperature and it's relevance from a non-linear system point of view in later chapters. Additionally, we define the softened softmax probability for a class $c$ for a particular output as in Equation 3.7.

$$p(c|x_i, \theta) = \frac{e^{x_i}/T}{\sum_j e^{x_j}/T} \tag{3.7}$$

It must also be noted here that the student logits, $o_z$, are also softened at the same temperature as $T$. A particular merit of this approach is that it is data agnostic. The teacher model could be trained on any data and the student model can be then trained on the same data, with a slightly lower accuracy, but a smaller model size. Moreover, once the student is trained, the teacher model is not required during inference. It is due to these merits, we chose to extend this work, and propose a more general loss function which will be introduced in Chapter 4.

### 3.4.2 Privileged Information and Knowledge Distillation

In [35], Vapnik, *et al*. propose the concept of learning using priviledged information or LUPI. The paradigm extends the traditional supervised learning paradigm where the pair $(x_i, y_i) \in \mathcal{D}$ are extended to a triplet $(x, x^*, y)$. Here $x^*$ is the *privileged* information *only* available to the teacher model at training time. For example, if we train a student network to segment different parts of a human tissue, then the images of tissue and associated labels will correspond to the dataset available to both student and teachers. However, the teachers also will have access to say, a medical report from a human doctor certifying if the tissue is healthy or not. This would be the privileged information.

In [26], Bottou *et al.* propose unifiying LUPI and Knowledge Distillation into one paradigm. The training algorithm is presented in Algorithm 4.

**Algorithm 4** Unifiying LUPI and Knowledge Distillation

1. Given a triplet $(x, x^*, y) \in \mathcal{D}_+$, separate $(x^*, y) \in \mathcal{D}_\mathcal{T}$ and $(x, y) \in \mathcal{D}_\mathcal{S}$.
2. Train a teacher model to learn $\mathcal{F}_\mathcal{T}$ with $\mathcal{D}_\mathcal{T}$.
3. Compute soft labels $s = \sigma(\mathcal{F}_\mathcal{T})$ for the student model by softening them at a temperature $T$.
4. Train the student model on $\mathcal{D}_\mathcal{S}$ with $s$ and $y$ using loss function from Equation 3.6.

In the algorithm, $\mathcal{D}_+$ denotes the overall training set comprising of privileged and common training samples. At the start, the set is partitioned into $\mathcal{D}_\mathcal{T}$ and $\mathcal{D}_\mathcal{S}$ where the teacher model is trained on the privileged information, and it generates the soft labels necessary for training the student. It is be very easily seen that in the case $x = x^*$, the paradigm becomes equal to the knowledge distillation paradigm.

A variation in this paradigm is provided by Zhang, *et al.* in [37]. Their proposed approach does not involve a highly trained teacher model. Instead, they propose training two or more students, where they learn from each other. Thus, the soft labels $s_i$ are the KL divergence between the output logits of a student and it's mutual learning partner. Equation 3.8 shows the loss function.

$$L1 = CE(o_{z1}, y_i) + KL(o_{z2}, o_{z1}) \tag{3.8}$$

Here, $CE(o_{z1}, y_i)$ is the cross entropy between one student's logits and the actual labels, and $KL(o1, o2)$ is the KL divergence. The loss is similar for the second student as well.

#### 3.4.2.1 Relation to Our Work

The aforementioned work proposes a paradigm that aims to distill knowledge from a specialized teacher to the student. Effectively, this aims to provide the student model some extra information that it may not learn if the teacher wasn't there to guide it. However, the paradigm makes no mention of learning from an ensemble. Moreover, knowledge transfer between a teacher and student is assumed to take place at a uniform temperature.

In the case of mutual learning, the authors measure the cross-entropy between the softmax proba-

bilities generated from both teacher and student models which is then used to train both the models at the same time. Zhang,*et al*. propose summing the KL-Divergences between each model in teacher ensemble and using that for the student, while broadcasting the student probabilities across all the models in the ensemble. However, this approach does not work since the KL divergence with student softmax proabilities offers lesser information to the teachers, leading to a poorer convergence, which in turn leads to student learning a bad representation of the classes.

Our approach in contrast, differs from the above mentioned approaches. We approach learning from multiple teachers where all teacher and student models are trained on $(x, y) \in \mathcal{D}$. However, we do not assume a uniform rate of knowledge transfer. In contrast with mutual learning [37], we propose a unidirectional flow from highly trained teacher models, thereby avoiding the problem of "bad learners" that has been previously mentioned.

## 3.5 Other Related Approaches

Recently, some other approaches to model compression (using knowledge distillation) have been proposed. Some of the approaches focus on training low precision student networks using different teacher models while other approaches focus on quantization and knowledge distillation. In this section, we briefly examine these techniques and discuss their differences with our method.

In Apprentice [27], Mishra, *et al*. propose a framework for trainining *low-precision* student networks with the knowledge distillation paradigm. A low-precision network would represent the weights by a signed `int` as opposed to a `float` or a `double` data type. This significantly reduces the memory consumption. The student (or apprentice) network is either trained alongside a full precision teacher model or knowledge is distilled from a trained full precision teacher model. Fine tuning a student model is also considered as another scheme. All three schemes expectedly show improvement in performance due to the distillation from a full precision trained model. However, the paper does not directly extend or modify the knowledge distillation paradigm itself and hence is not directly related to our work.

An approach to unifying quantization and distillation is proposed by Polino, *et al*. in [30] where

they propose *quantized distillation*. The algorithm describes a way to use quantized weights in a student model and then use the knowledge distillation paradigm to learn better connectivity. A way to visualize the input to this scheme is to consider $Q = \{q_1, q_2, ...q_n\}$ discrete bins that contain quantized weights. The continuous weights $w_{ij}$ belonging to a certain $q_i \in Q$ can be computed using a quantization function that performs a mapping from a continuous to discrete domain. Next, if $W_Q$ be the set of quantized weights obtained from a quantization function, then distillation loss can be computed w.r.t. $w_q \in W_Q$. However, during backward pass we need to somehow update $w_{ij}$. Polino, *et al*. propose updating the weights in *full precision* before *requantizing* the weights. We observe that the quantization function is not differentiable for every input i.e. $\frac{\partial L_q}{\partial w_q}$ only exists when $w_q$ belongs to a particular $q^{th}$ bin, it is zero everywhere else. Moreover, the quantize-train-quantize cycle has a computational bottleneck of quantizing the weights during the forward and backward pass in one epoch. To get around this limitation, the authors propose *differentiable quantization* in which they define the gradient of the quantization function and eliminate the need for re-quantizing. Our work in contrast, does not need quantization to perform well. A major design goal of our approach is to eliminate the need for workarounds in the deep learning train-eval-deploy cycle. A student model trained via our approach can be effectively considered a smaller instance of a larger ensemble and expected to perform just as well as the ensemble would.

Finally, we discuss a recent work by Lan, *et al*. in [23] where they propose an on the fly ensemble as shown in Figure 3.6. The idea is to extend a single teacher model with a common shared low level layer and $m$ auxilary branches. Thus, the teacher is an ensemble of $m$ branches which learn the same low level representation but higher order features are learned differently by each $1, 2...m$ branches. It must be noted that the auxiliary branches cannot differ too much from the base lower level, since it will make gradient propagation to the lower layers difficult. For example, if a branch $i$ resembled VGG-11 and branch $j$ resembled VGG-19 then it is possible that gradients from $j^{th}$ branch may never be backpropagted back to the lower layers. The "on-the-fly" part of this approach comes from the fact that $m$ is variable and hence it can be adjusted at run time.

Overall, the $m$ auxiliary branches are treated as an ensemble of teachers and each branch is a student model. The logits from all branches are collected via a gate module and then knowledge is distilled back into each individual branches. The temperatures however are kept uniform. Our

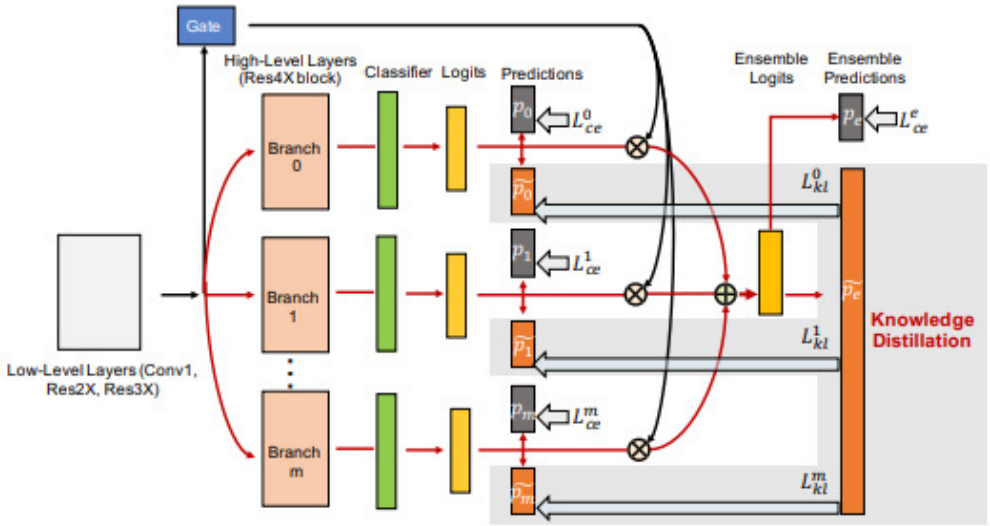


Figure 3.6: Knowledge Distillation by On the fly Ensemble (ONE) by Lan *et al.*

approach has some similarity to this work since we consider learning from an ensemble. However, there are a couple of differences. The learning approach proposed by Lan, *et al.* can be considered an extended case of *mututal learning* [37] where the ensemble of $m$ branches is mutually trained with each individual branch. In contrast, our approach is to take a pre-trained ensemble of different teacher architectures and *non-uniformly* distill knowledge from them into the student. Next, we note that the architecture of the auxiliary branches can not be too distant from the architecture of the lower layers,- this limits the overall choice of ensemble to a particular architecture. However, in our proposed approach there is no limitation on the architecture of the student model since we assume that the teachers are not trained during the learning phase.

## 3.6 Chapter Summary

In this chapter, we defined the problem of model compression by showing the energy and memory consumption of deep neural networks. We then examined related approaches starting from the earliest work to the current state of the art. Additionally, we compared the approaches to our proposed algorithm. In the next chapter we formally introduce our specific contribution.

# Chapter 4

# Experience Loss

In the previous chapters we have introduced the underlying framework for supervised learning tasks and looked at training algorithms to train Deep Neural Networks. We have also motivated the problem of model compression and examined in some detail the different approaches that have been proposed so far along with their relative benefits and demerits. In this section, we introduce our specific contribution. Section 4.1 begins by formalizing the notation used throughout the analysis in this Chapter. We introduce the proposed loss function in Section 4.2 and theoretically analyze the gradient to show that we can control the knowledge distilled by varying a single hyperparameter. Section 4.3 then introduces a memory efficient training scheme that is designed to scale well with different number of teachers. Finally, in Section 4.4 we look at two student architectures that are designed for memory efficiency. We survey existing approaches and propose a new architecture.

## 4.1 Notation

Let $(x_i, y_i)$ denote the input and label sample pairs of a training set $\mathcal{D}$. Also let $E = \{m_0, m_1..m_{N-1}\}$ be a collection of $N$ deep neural networks that have high accuracy on $\mathcal{D}$. For a particular input sample $x_i \in D$ we obtain a softmax response $o$ from the model. This vectoris of length $C$ representing the probabilities of $x_i$ belonging to a class $c_i \in C$. For a model $m_i$ we can define the softened-softmax for a particular class $y_j \in C$ at a temperature $T_i$ as:

$$o^{T_i} = \frac{e^{y_j/T_i}}{\sum_z e^{y_z}/T_i} \tag{4.1}$$

For a particular output class $j$, we define the set $\kappa = \{o^{T_0}, o^{T_1}..o^{T_{N-1}}\}$ to be the softened softmax response at different temperatures. In other words, $k_i \in \kappa$ represents the soft-softmax repsonse at temperature $T_i$.

## 4.2 Experience Loss

Our goal is to train an optimal smaller student model that learns a function $\mathcal{F}_{\mathcal{S}}$ under the influence of teachers in $E$ and the labels $y_i \in \mathcal{D}$. We hypothesize that the student model performs much better when distillation occurs from multiple teachers at different temperatures and the training process is more stable due to the regularization effect from different teacher architectures. The former intuition arises due to the observation that an ensemble of models trained on the same task have much better performance than a single model and the latter intuition arises by noting that the student model is trained under the influence of teacher architectures that have obtained a minima over the input space. Thus, the output of such teachers act as regularizers to the student model. We mathematically define our approach as:

$$L_{exp} = \alpha \sum_{\substack{i=1 \\ k_i \in \kappa}}^{N} KL(o^{T_z}, k_i) + (1-\alpha)\mathcal{H}(\tilde{y_z}, y) \tag{4.2}$$

The first term in Equation 4.2 is the KL-divergence between the softened-softmax of the student $o^{T_z}$ and the softened-softmax of the teacher models for a particular class ($c_j \in C$). The student logits are softened at a temperature $T_z$ using Equation 4.1. We will discuss different approaches for initializing $T_z$ in the experiments section. For the purpose of the analysis, we will consider $T_z$ to be an independent quantity that can be adjusted independently. The second term in the loss function is the cross-entropy between the student's unsoftened softmax prediction and the acutal labels for a given input sample. Both terms are weighted by a hyperparameter $\alpha$ that controls the weight we put on each term's contribution. Throughout this work, we will weigh the first term more heavily than the second to encourage the student to learn more from the teacher models.

### 4.2.1 Novelty and Significance

We have discussed different approaches to knowledge distillation in Chapter 3. While Hinton *et al*. [13] discuss the possibility of extending their approach to an ensemble of networks, they do not provide a theoretical framework for the same. The novelty of our work lies in the fact that we are the first to consider a diverse ensemble of deep neural networks and distill knowledge into a smaller model. We do so without employing any bagging or boosting techniques and experimentally show that a simplistic formulation yields better results than in [13]. Additionally, our framework allows for a non-uniform rate of knowledge transfer, even though our formulation relies on pre-selection of these rates. To the best of our knowledge, we are the first to consider the problem of knowledge distillation from the point of non-uniform rate of transfer.

Another contribution of our work is to define an end-to-end training algorithm that can be memory-efficient at training time by querying the ensemble for their outputs on a particular model. This approach can be further adapted by querying only the top $k$ most important models, with the importance being determined by a small meta-network or a nearest neighbor analysis.

### 4.2.2 Theoretical Analysis

In order to understand what the student network learns from an ensemble $E$, we look at the gradient of our loss function w.r.t. and output logit $z_j$. This logit is produced by a student model in response to an input sample $x_i$. We assume that for the same input sample a model $m_i$ produces a logit $v_{ij}$ and a softened softmax response $k_i$. The error contribution of the logit w.r.t to the loss function can be defined as:

$$\frac{\partial L_{exp}}{\partial z_j} = \alpha \frac{1}{T_z} \sum_{\substack{i=1 \\ k_i \in \kappa}}^{N} (o^{T_z} - k_i) \tag{4.3}$$

We use the definition of softened softmax from Equation 4.1 to expand Equation 4.3 as:

$$\frac{\partial L_{exp}}{\partial z_j} = \alpha \frac{1}{T_z} \sum_{i=1}^{N} \left( \frac{e^{\frac{z_j}{T_z}}}{\sum_c e^{z_c}/T_z} - \frac{e^{\frac{v_{ij}}{T_i}}}{\sum_c e^{v_{ic}}/T_i} \right) \tag{4.4}$$

Here $T_z$ is the student temperature, i.e. the temperature at which the student logits are softened and $c$ is an index into all possible classes $C$ for the given data set. If $T_i >> v_{ij}$ then we can approximate

Equation 4.4 via Taylor Series Expansion and re-write it as:

$$\frac{\partial L_{exp}}{\partial z_j} = \alpha \frac{1}{T_z} \sum_{i=1}^{N} \left( \frac{1 + \frac{z_j}{T_z}}{C + \frac{1}{T_z} \sum_c z_c} - \frac{1 + \frac{v_{ij}}{T_i}}{C + \frac{1}{T_i} \sum_c v_{ic}} \right) \tag{4.5}$$

Equation 4.5 shows the gradient of the first term of experience loss with no approximations and $M$ possible output classes. We can further simplify this equation by considering that we usually zero-mean the input training examples (i.e. subtract the mean from each of the individual samples), thus we can consider $\sum_c z_c$ and $\sum_c v_{ic}$ to be zero. Hence, we can approximate the gradient as:

$$\frac{\partial L_{exp}}{\partial z_j} \approx \alpha \frac{1}{C T_z} \sum_{i=1}^{N} \left( \frac{z_j}{T_z} - \frac{v_{ij}}{T_i} \right) \tag{4.6}$$

From Equation 4.6 we observe that the contribution of the KL divergence w.r.t. a student logit $z_j$ is effectively the sum of the *difference* between the softened logit and the the softened logits of the teachers in the ensemble $E$. A key distinguishing feature between our formulation and the one proposed by Hinton *et al.* in [13] is that we assume that $T_z$ and $T_i$ can be adjusted independently. The distillation from a particular teacher can then be controlled by varying $T_i$. Interestingly, if we consider $T_i = T = T_z$ and $N = 1$, Equation 4.6 reduces to:

$$\frac{\partial L_{exp}}{\partial z_j} \approx \alpha \frac{1}{C T^2} [z_j - v_j] \tag{4.7}$$

Equation 4.7 is similar to the one proposed by Hinton *et al.* in Distillation Loss [13]. We can easily see how our approach, which we call Experience Loss, describes a more general case of Knowledge Distillation. We shall later show this experimentally that Experience Loss also outperforms Distillation Loss if the temperatures are chosen correctly. Additionally, we will showcase some "strategies" for initializing $T_z$ from $T = \{T_1, T_2..T_N\}$.

## 4.3 Training Procedure

In this section we describe a training algorithm that allows for training a student model with multiple teachers. Our goals are twofold - we want the student model to be at least as accurate as the teacher models (within 5% tolerance limit) and the training to be scalable as different teacher models are added to the ensemble. Hence, we focus on *memory efficiency* as opposed to the *compute cost* of the logits.

**Algorithm 5** Experience Loss Training Procedure

1. Select a training set $D$ containing $(x, y)$ pairs where $x \in \mathbb{R}^{hxwxc}$ and $y \in \mathbb{C} = \{1..m\}$.

2. Select set of teacher models $E = \{E_1..E_N\}$ and instantiate student $S$.

3. Select the teacher temperatures $T$ for teachers and set initialization strategy for $T_z$.

4. Compute the forward pass

**procedure** FORWARD PASS($D, E, S, T, T_z$) ▷ Compute $L_{exp}$
  **while** $(x_i, y_i) \in D$ **do**
    $l_{kl} \leftarrow []$
    $s \leftarrow S(x_i)$
    $i \leftarrow 0$
    **while** $i \neq N$ **do**
      $t_{out} \leftarrow E_i(x_i)$
      $l_{kl} \leftarrow KL(s/T_z, t_{out}/T_i)$
    **end while**
    $l_{ce} \leftarrow CE(s, y_i)$
    $L_{exp} \leftarrow \alpha * \sum(l_{kl}) + (1 - \alpha) * l_{ce}$
  **end while**
**end procedure**

5. Compute backward pass using SGD or similar optimizer.

In Algorithm 5 we see that the training procedure is designed to be as network and dataset agnostic as possible. The inputs to the algorithm is a dataset $\mathcal{D}$ containing $x_i \in R^{hxwxc}$ and $y_i \in \{1..m\}$, a student network $S$, and an ensemble $E$. The hyperparameters are $\alpha$, $T$, and $T_z$. The main procedure is described in `Forward Pass`. For one epoch, the procedure initializes $l_{kl}$ and then computes the student and teacher logits. The KL divergences are added to $l_{kl}$ and finally the loss is obtained as a weighted sum.

The key to the memory efficient procedure is the inner most `while` loop. We see that we obtain $t_{out}$ by passing the input training feature $x_i$ through a network $E_i \in E$. This happens at *run time* instead of pre-caching of the logits. Pre-caching of logits is a major memory bottleneck and severely limits the amount of teachers a student model can be trained with at a given time. Our approach however, simply queries the outputs at run time and the result is detached from the computational graph i.e. the gradient of $L_{exp}$ is not backpropagated to the teacher models. We found the machine learning library `PyTorch` [29] to support dynamic computational graphs and implemented this training procedure in Python using this library.

We have mentioned that the loss function is made up of two functions that are differentiable everywhere. We take advantage of this in our training procedure and compute the gradients of the two terms separately and backpropagate through different layers. This approach also allows us to use different `optimizers` like SGD, Adam [19], etc. in a plug and play fashion. Overall, the training procedure is highly modular and can work with different datasets, network architectures, and teacher ensembles.

## 4.4 Student Network Architectures

The knowledge distillation paradigm is flexible when it comes to the architectures of teachers and students. However, the effectiveness of the approach is limited to some degree by the student architecture used during training. Another factor is that the student model is ultimately intended to be deployed on a resource constrained device, which means that the student architecture has to as memory efficient as possible. In this section, we will begin by discussing some prior work in the areas of memory-efficient architectures and then introduce our contribution in the form of memory efficient student architectures that directly benefit from the experience of an ensemble of teachers.

### 4.4.1 Prior Work

Network architecture design is a broad space and design decisions are often trade-offs between a network's size and the generalization ability. Additionally, the target application influences the decisions made in different stages of the network. We discuss some related work in the areas of creating small and efficient network architectures that can be deployed on devices with memory constraints.

The 3-D Convolutional operation found in most of the modern CNNs is a parameter heavy operation. Algorithmically, a Convlayer $C$ takes in an input feature map of dimensions $(H \times W \times C)$ and produces $N$ output feature maps of dimensions $(H^{'} \times W^{'})$. A convolutional filter $F$ has dimensions $F_H \times F_W \times C$. Usually $F_H$ and $F_W$ are equal and have odd number sizes to produce appropriate feature maps. Overall, for $N$ such filters, a convolutional layer has $N \times F_H \times F_W$ parameters. If $F_H, F_W > 1$, then this becomes a parameter heavy operation. In [15], Iandola *et al.*introduce SqueezeNet, i.e. a network architecture that has similar accuracy to that of AlexNet but occupies much less memory. A key design decision they introduce is to replace the majority of $(3 \times 3)$ filters

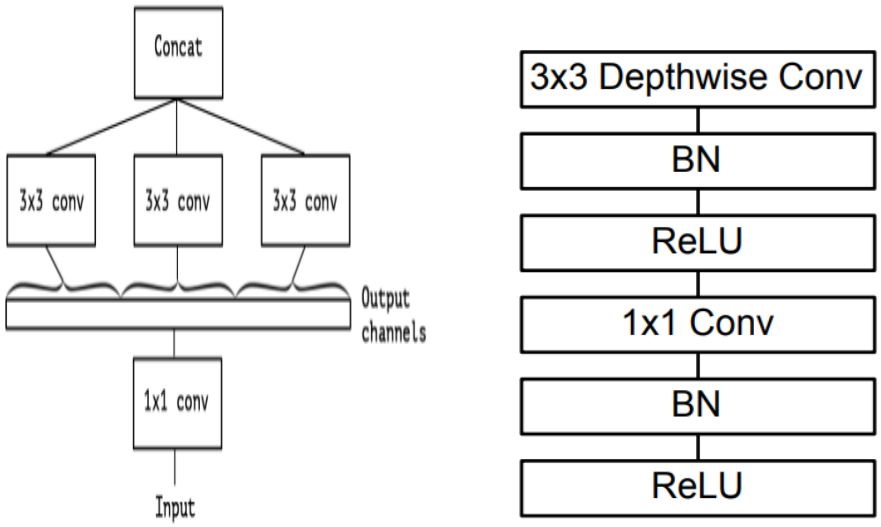


Figure 4.1: Reduced Compuation Architectures. Left: The Xception Module, Right: The MobileNet module.

(i.e. $F_H = F_W = 3$) with $(1 \times 1)$ filters. Another important observation they make is to reduce the input channels to $(3 \times 3)$ convolution filters.

Another important observation about convolutional layers is that they are computationally intensive. As reported in [14], a regular convolutional operation is $O(F_H \times F_W \times N \times C \times H \times W)$. However, the convolutional operation can be made parallel. We consider a 3-D convolution operation to be a spatial correlation of features across an input image and then a linear cross-correlation across channels. However, these two operations can be separated [14, 4] into two specific operations - DepthWise Convolution and Point Wise Convolution. The former is a spatial convolution that operates on the spatial dimension $F_H x F_W$ in size while the latter is a channel wise convolution that operates on the channel dimensions to accept $M$ channels as input and produce $N$ channels as output. Moreover, the order of operations can be interchanged. Given this formulation from [14], the reduction in computation is:

$$\rho = \frac{1}{N} + \frac{1}{F_W * F_H} \tag{4.8}$$

Based on this observation, two architectures have been proposed. Figure 4.1 shows the Xception [4] and Mobilenet [14] building blocks. In the former, the input is converted to an $N$ channel out-

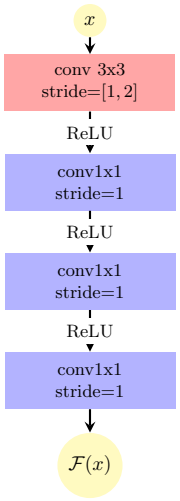


Figure 4.2: The Wide Basic Block (Wbv1) Architecure.

put, which is then spatially convolved with $(3 \times 3)$ filters. The result is then concatenated. If we denote $\mathcal{F}(F_H, F_W, N)$ to be the original spatial 3-D convolution then the Xception block learns $\mathcal{F}_1(1, 1, N) + \mathcal{F}_2(F_H, F_W, 1)$. Similarly, the mobilnet module learns the same represenation with a small difference - after every depthwise/pointwise convolution the output feature map is passed through a nonlinearity. This approach is different from the former approach where the non-linearity is applied to the output of $\mathcal{F}_1 + \mathcal{F}_2$.

Many newer architectures that are optimized for mobile applications, employ residual connections. Most notably in [33] Sandler, *et al*. introduce the concept of inverted residuals and linear bottlenecks. These two are relatively new techniques and beyond the scope of discussion in this work. We however mention residual connections because in [36] Zagoruyko, *et al*. make an interesting observation about ResNets - the standard residual branch often ends up learning small or no represenation of the input at all. They then introduce the WideResNet architecture, which adds a parameter heavy residual branch and outperforms the conventional ResNet in classification tasks.

### 4.4.2 The WideShallow and WideDeep Architectures

In this section we build off of these ideas to present two types of student architectures used in studying the effects of our proposed loss function. The architectures are motivated by initial studies that used 5 or 6 layer networks. These networks, though small in terms of memory consumption,

| Type | Filter Shape | Input Shape | Output Shape | # Params |
|---|---|---|---|---|
| conv1, stride=2 | 3x3x32 | 32x32x3 | 17x17x32 | 896 |
| WBv1, stride=1 | 3x3x32x64 | 17x17x32 | 17x17x64 | $31.23K$ |
| max_pool | - | 17x17x64 | 6x6x64 | 0 |
| WBv1, padding=1, stride=1 | 3x3x64x128 | 6x6x64 | 8x8x128 | $123.90K$ |
| batch_norm | - | 8x8x128 | 8x8x128 | 256 |
| avg_pool 4x4 | 4x4 | 8x8x128 | 2x2x128 | 0 |
| Linear | 512x1 | 1x512 | 1x$M$ | 5130 |
| `Total Params` | - | | | $161.48K$ |

Table 4.1: The Wide Shallow Net architecture. In the second wide basic block padding is applied for correct input propagation. Avg Pooling is performed instead of Max Pooling in upper layers.

were unable to converge under the influence of different teacher models. Thus, we reviewed existing literature and developed two newer architectures to study the effects of non-uniform temperature distillation. A common feature in both of the architectures is the use of convolutional blocks that learn a particular $\mathcal{F}(x)$. These blocks are then replicated at different stages in the architecture. However, the design of these blocks are different in both architectures. We shall look at the two architectures separately.

#### 4.4.2.1 Wide Shallow Architecture

Figure 4.2 shows the architecture of the WideBasic (WBv1) block. The block is composed of stacks of 4 layers, the first of which is a $3 \times 3$ convolution; the rest are $1 \times 1$ convolutions. This structure of the block follows from Iandola, *et al.* observation in [15], where replacing majority of the $3 \times 3$ convolutions with $1 \times 1$ depth convolution resulted in significantly less parameters with no significant impact on accuracy. The first convolutional layer is necessarily "wide" to capture more regions of interest in a given training input pattern. Table 4.1 shows the overall output shapes and the parameter in the Wide Shallow Architecture.

The shallow architecture has $161.48K$ parameters. If we assume full precision weights, then the size of the network comes to about $2MiB$ after compression by a standard utility like tar or zip. We also note the lack of aggressive max-pooling operations in the architecture since it has been observed that max-pooling leads to a loss of spatio temporal information. Our proposed architecture is not too deep and hence it cannot afford to lose the spatio temporal information across the feature maps. Following Iandola, *et al.* [15], we replace max-pooling with average pooling before flattening the

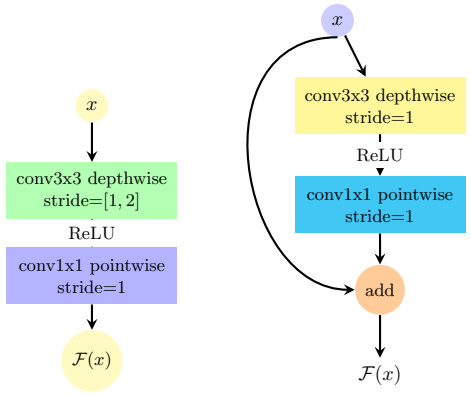


Figure 4.3: The WideBasicv2 (WBv2) blocks. Left: The entry/exit blocks; Right: The middle residual bock.

feature vector. The reason for this is that the activations produced by a depth convolutions are not high in magnitude. A max-pooling operation would cause a loss of information and consequently lead to worse performance on a given dataset.

Although not shown in Figure 4.2, each convolutional layer in the stack is followed by a `BatchNorm` [17] layer.

#### 4.4.2.2 Wide Deep Architecture

Figure 4.3 shows the newer version of the WideBasic block introduced in the Wide Shallow architecture i.e WideBasicv2 (WBv2). This block incorporates some of newer approaches proposed in recent literature [14, 4, 12, 36]. More specifically, it introduces separable convolutions and residual connections in parts of the network. We have already discussed the benefits of separating a convolution operation into depthwise and pointwise convolution operations in Section 4.4.1.

We borrow the concept of "network flows" from Chollet *et al.* [4] and divide the network into three flows - entry, exit and middle. For the entry and exit layers, the "WideBasic" block is a depthwise separated convolution block with `ReLU` and `BatchNorm` [17] layers in between. This approach is similar to Mobilenetv1's [14] block. The stride of the $3 \times 3$ could be set to $x \in [1, 2]$. Since we do not use maxpooling operation in the architecture, we use strided convolutions to reduce the spatial dimensions of the feature map. The middle flow is similar to entry/exit flows in the sense that it uses depthwise separable convolutions. However, it restricts the 3x3 depth wise layer to have a stride of

| Type | Filter Shape | Input Shape | Output Shape | Parameters |
|---|---|---|---|---|
| 3x3 Conv stride=2 | 3x3x32 | 32x32x3 | 16x16x32 | 928 |
| WBv2, stride=1 | (3x3) + (1x1)x32 x 64 | 16x16x32 | 16x16x64 | 2528 |
| WBv2, stride=2 | (3x3) + (1x1)x64 x 128 | 16x16x64 | 8x8x128 | 9152 |
| 3x WBv2, stride=1 | (3x3) + (1x1)x128x256 | 8x8x128 | 8x8x256 | 34688 |
| WBv2, stride=2 | (3x3) + (1x1)x256x512 | 8x8x256 | 4x4x512 | 206.59K |
| 1x1 conv, stride=1 | 1x1x512x8192 | 4x4x512 | 4x4x8192 | 4194.30K |
| AvgPool 4x4 | 4x4 | 4x4x8192 | 1x1x8192 | 0 |
| Linear output=$M$ | 8192x1 | 1x8192 | 1x$M$ | 81.9K |
| Total | | | | 4.53M |

Table 4.2: The Wide Deep Net Architecture. WBv2 refers to the newer version of the wide basic block.

1 and the input/output channels to be the same. Additionally, it introduces residual mapping from the input. It must be noted that we do not follow Zagoruyko, *et al*. [36] and keep the residual branch free of any trainable parameters since our objective is not to attain state of the art performance but to have memory efficient architectures.

Table 4.2 shows the overall architecture of the proposed Wide Deep Net. The entry and exit blocks are organized as shown in the left side of Figure 4.3 and the middle ones are arranged as on the right of the same figure. The first two `WBv2` blocks constitute the entry flow while the last `WBv2` block forms the exit flow. Since the newer wide basic blocks have separable convolutions the filter sizes are shown separately. We note that we do not use any sort of max-pooling operations in this architecture and restrict ourselves to one average pooling layer before the feature maps are passed to the logistic regression branch. The inputs to the architecture are assumed to be of $x \in \mathbb{R}^{32\times32\times3}$. Additionally, we introduce a $1 \times 1$ convolutional layer to produce a "flat" output feature map. This is later down sampled by the average pooling layer.

We immediately see that this network has more trainable parameters and consequently bigger size than the Wide-Shallow Architecture. However, the majority of the trainable parameters are contributed by 1x1 convolutions. Thus they do not end up occupying too much memory (even when full precision is used). The total size of this networks comes to about 18-24 Mib, without employing any file compression algorithm.

## 4.5 Chapter Summary

In this chapter we derived our approach to model compression by knowledge distillation. We then presented a training procedure to train with an ensemble of teacher models by querying them for output during the run time. Furthermore, we introduced two memory efficient architectures by building off of ideas found in literature. We look at experiments and the results on these architectures with our proposed loss function in the next chapter.

# Chapter 5

# Experiments and Results

This chapter introduces the results of our experimental simulations with the two student architecture introduced in the last chapter on different datasets. We group our experiments by architecture since we find that the student models learn differently. Section 5.1 details the results with the wide-shallow student architecture and Section 5.2 details the results with the wide-deep architecture.

## 5.1 Experiments with Wide Shallow Net

This section introduces the experimental setup and results with different teacher models with the wide-shallow network(Section 4.4.2) as the student model. To draw better conclusions, we kept the dataset as an invariant. However, we perform different experiments with other datasets, the results of which are discussed in future sections.

CIFAR-10 [21] is a dataset by Krizhevsky *et al*.containing 60,000 images and 10 classes. The dataset is further subdivided into training set $\mathcal{D}_{train}$ containing 50,000 images and a test set $\mathcal{D}_{test}$ containing 10,000 images. All images are of dimensions $\mathbb{R}^{32x32x3}$. Figure 5.1 shows some images in the dataset.

### 5.1.1 Experimental Setup

To train the wide-shallow net we trained VGG-11, VGG-19 [34], WideResNet [36] and ResNet-18,50,110 [12] as the teachers. All teachers were trained for 200 epochs with a learning rate of $1e^{-2}$.

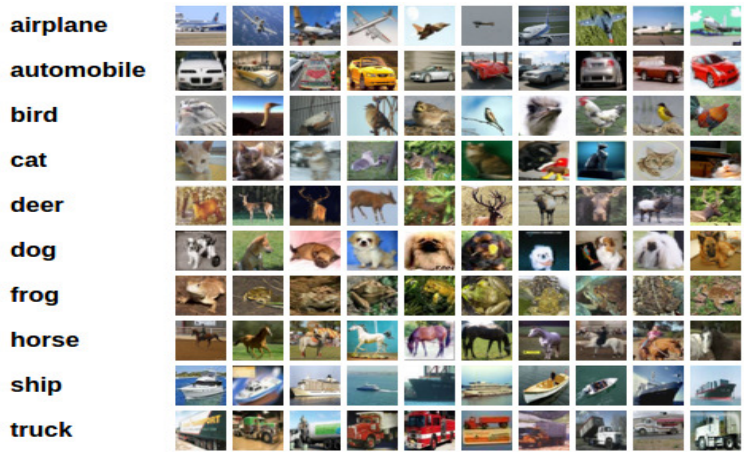


Figure 5.1: A sample of images in the CIFAR-10 Dataset. (Krizhevsky *et al.*)

The student model in turn was trained for 50 epochs with the same learning rate. In our initial experiments we found that if the Adam optimizer was used in training the student model, it would lead the student model to overfit the training set. We surmise that since Adam is an adaptive algorithm, shallow networks may quickly shoot beyond the local minima. On the other hand an SGD based optimizer, works according to Algorithm 1 and thus is dependent on the current state of the gradient. We noted that the training was more stable, and the trained models did not overfit the data. Moreover, we noted that training with multiple teachers has a regularizing effect on the student model. We will discuss this effect in more detail in the coming sections.

To provide an effective comparison between Distillation Loss [13] and our proposed approach, we use the distillation loss formulation in cases when there is one teacher or $(T_i = T) \forall T_i \in T$. In every other case, we use Experience Loss (Equation 4.2) to compute the loss.

### 5.1.2 Initialization Strategies and Optimal Temperature

We have shown in Equation 4.6 that we can control the amount of knowledge distilled from a teacher to a student by independently varying $T$ and $T_z$. Since we treat the softening temperatures as hyperparameters, we need to choose optimal values for a given task and dataset for both teacher and student temperatures which introduces a lot of manual tuning in the overall framework. One way to reduce manual tuning could be to auto-initialize the student temperatures from given teacher temperatures. This constrains all manual tuning to finding optimal teacher temperatures. In this

work, we introduce three initialization strategies:

- Mean Strategy: Initialize the student temperature to be the mean of teachers, i.e. $T_z = mean(T)$.
- Max Strategy: Initialize the student temperature to be the max of the teachers, i.e. $T_z = max(T)$.
- Min Strategy: Initialize the student temperature to be the min of the teachers, i.e. $T_z = min(T)$.

To study the effects of temperature on the knowledge distillation we performed controlled experiments with temperatures in the range $[5, 50]$ with a step size of 5. In the case of two or more teachers, we empirically found that the networks converged well when $T < 30$. As temperature is increased, we found that the accuracy dropped uniformly in all initialization schemes. We also tried combining very high and very low temperatures and observed that the network either did not converge or it learned very little from the teachers. Table 5.1 shows the results of our experiments with different number of teachers in the ensemble.

| Teachers | Mean | Max | Min |
|---|---|---|---|
| VGG-11 | 0.171 | 0.174 | 0.18 |
| VGG-19, VGG-11 | 0.169 | 0.163 | **0.161** |
| Resnet18, VGG-19, VGG-11 | 0.172 | 0.168 | **0.166** |
| Baseline | 0.207 | | |

Table 5.1: Top-1 error rates on CIFAR-10 validation set with different temperature initialization schemes and teachers. The baseline was trained without using any teachers.

To measure the effects of temperatures, we performed an ablation study by progressively adding heavier architectures. We distinguish a "heavy" architecture by noting that it has significantly more trainable parameters than a "light" architecture. In the case of one teacher, where we use Distillation Loss, we note that the error rate is uniformly worse than the ones obtained with Experience Loss and two/three teacher ensembles. In the case of a three teacher ensemble, we note that the error rates are slightly higher than the two teacher ensemble. However, we note during training that the three teacher ensembles produce a much more regularizing effect than two teacher ensembles. Figure 5.2 shows the accuracy on the test set for three initialization strategies for both two and three teacher ensembles.

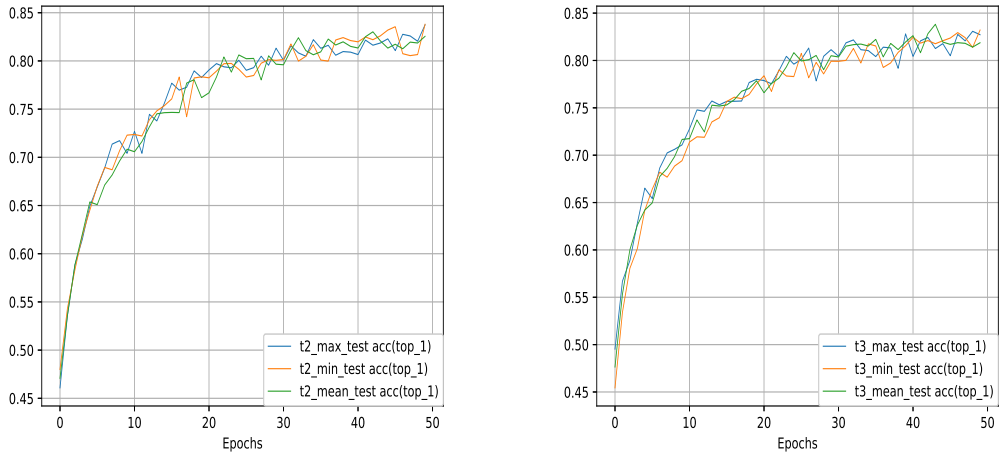


Figure 5.2: Relative comparison of accuracies of two and three teacher ensembles.

The left graph shows the validation accuracy with two teacher ensemble and the right graph shows the same with three teacher ensemble. We note that for the three teacher ensembles, the graph is a lot smoother and as training progresses there are lesser random spikes, showing that the student is better regularized. We also note that well trained teachers can also be used as regularizers for more complex networks.

### 5.1.3 Effect of Teacher Ensembles

In our experiments with different temperature initialization schemes we noted that softening different teacher models at different temperatures could produce different results. In order to study the relative effect of different teacher models on overall knowledge distillation, we performed experiments with different combinations of teachers. For ease of understanding, we encode the ensembles as follows:

- $E1$: Two teacher ensemble consisting of VGG-19 and Wide-ResNet.
- $E2$: Two teacher ensemble consisting of Resnet-18 and Wide-ResNet.
- $E3$: Two teacher ensemble consisting of VGG-11 and Wide-Resnet.
- $E4$: Three teacher ensemble consisting of VGG-11, VGG-19 and Wide-Resnet.

| $E1$ | | | $E2$ | | | $E3$ | | | $E4$ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| mean | max | min | mean | max | min | mean | max | min | mean | max | min |
| 83.10 | 82.57 | 83.12 | 82.58 | 83.10 | 83.52 | 83.67 | 83.21 | 83.27 | 82.92 | 83.41 | 83.46 |
| 83.84 | 83.69 | 83.33 | 83.84 | 83.69 | 83.33 | 83.84 | 83.69 | 83.33 | 83.03 | 83.53 | 83.60 |

Table 5.2: Top-1 Validation accuracies(%) with different ensembles and different initialization schemes.

Based on our observations from the optimal temperature selections we constrain the teacher temperatures to be $T \in \{10, 15, 30\}$. We set the softening temperature for heavy architectures such Wide-Resnet and VGG-19 to be 15 and lighter architectures like VGG-11 to be 30. In the case of three-teacher ensemble, we use all three temperatures. Empirically, we found that these settings allowed a student to converge to a much better local optimum. Table 5.2 shows the results of our experiments.

The top row in Table 5.2 shows the top-1 validation accuracy with different ensembles. The bottom row, shows the top-1 validation accuracy obtained with the ensembles in the optimal temperature experiments (Section 5.1.2). We make several observations from the table. When two heavy architectures are paired together ($E1$, $E2$), then accuracy uniformly dips across all three initialization schemes. Conversely, we find that two heavy architectures (WideResNet and ResNet-18) perform relatively well when the "min" initialization scheme is used. For the "mean" initialization strategy and two teacher ensemble, we find the student model learns best with a combination of two lighter architectures. Additionally, we see that when a heavier architecture is paired with lighter one ($E3$), the accuracy drops a little. Interestingly, the "min" strategy is the least affected by the change in ensembles. In fact, in $E2$, the performance even improves marginally. Since the gradient of Experience Loss (from Equation 4.6) is the sum of differences between two softened logits, we surmise that softening the student logits by different initialization schemes, gives rise to varying magnitude of the *difference* between the two logits. With the "min" strategy, the logits of the student are softened very little, giving a strong indication that lower temperatures could lead to better knowledge transfer. Figure 5.3 shows the error rates of the two teacher ensemble using two different initialization schemes.

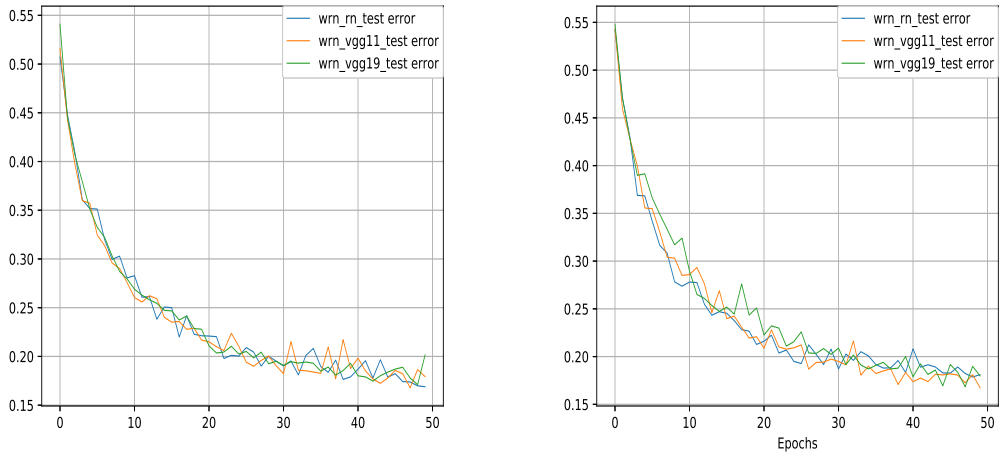


Figure 5.3: Error Rate on CIFAR-10 validation with max and min strategy. Left: Max Strategy, Right:Min Strategy.

## 5.2 Experiments with Wide-Deep Net

In this section, we present our experiments with the Wide-Deep architecture. Like the Wide-Shallow architecture, we present our results in two different types of studies. The first study focuses on the effect of varying the temperatures with different number of teachers in the ensemble. The second study looks at the effect of varying the network architectures of the teachers in a given ensemble. As with the Wide-Shallow net, we present our results on the CIFAR-10 dataset. The rationale behind keeping the dataset constant in the studies is that it allows us to isolate the effect of different temperatures and ensembles.

### 5.2.1 Teacher Ensemble Study

In our earlier study with Wide-Shallow network we noted that the final accuracy of the student model depended on the architecture of the teacher models in a given ensemble. We perform the same study with the Wide-Deep Net with a more diverse set of teacher ensembles. We combine VGG-11, VGG-19 [34], Resnet [12] and WideResNet [36] in different combinations to study the relative effects of mixing different architecture styles. We encode these ensembles as follows:

- $E_1$: VGG-19, WideResNet
- $E_2$: ResNet-18, WideResNet

| Strategy | E1 | E2 | E3 | E4 | E5 |
|---|---|---|---|---|---|
| Mean | 83.15 | 83.18 | 84.27 | 83.71 | 83.73 |
| Min | 84.05 | 84.21 | 83.73 | 83.87 | 84 |
| Max | 83.61 | 83.22 | 83.48 | 83.56 | 84.29 |

Table 5.3: Top-1 Validation Accuracies(%) with different ensembles and initialization strategies

- $E_3$: VGG-11, WideResNet
- $E_4$: VGG-11, VGG-19, WideResNet
- $E_5$: VGG-11, VGG-19, ResNet-18

For each of these ensembles we compared the effect of the three initialization strategies mentioned in Section 5.1.2 which allows us to draw conclusions about the best strategy for initialization given a fixed teacher ensemble. These heuristics work in the same way as selecting hyperparameters and can be adaptively used during the training process. Table 5.3 shows the results of our experiments with these ensembles. All experiments are carried out for 50 epochs with a learning rate of 0.01 on CIFAR-10 dataset. For the two teacher ensembles the temperatures were set to $T = \{5, 10\}$ and the three teacher ensemble were set to $T = \{5, 10, 15\}$.

We make some interesting conclusions from the table. First, we observe that like Wide-shallow net, the accuracy of Wide-Deep Net also decreases uniformly across all initialization schemes. This consistent observation lends strength to our original argument that adding more number of teachers creates a sort of regularising effect which may not lead to increase in the accuracy but allows for more stable training. Second, we observe that for the "min" strategy the accuracy increases uniformly with a "heavy" ensemble. This accuracy however drops noticeably when a "light" architecture is paired with a "heavy" architecture. Since the Wide-Deep Net is heavier than architectures like VGG-11 and has a higher baseline accuracy than VGG class of models, setting a lower temperature for a lighter architecture is akin to encouraging the student from a low-accuracy model. Interestingly, the accuracy for the "mean" initialization strategy *increases* when a lighter architecture is paired with a heavy one. Additionally, we observe that the "min" initialization strategy works best across different ensembles. This observation can be explained by Equation 4.6 and expanding the sum where $T_z = min(T)$. The sum is necessarily bigger for logits from heavy architectures and smaller with lighter architectures. The overall sum then provides more information to the student

model, allowing it to learn a much more complete representation. When compared with mean and max strategies, the sum of differences is much higher initially and converges to a lower value in the end.

For the case with three teacher ensembles, we note that the accuracies with different initialization schemes are more closely clustered together. Even so, we notice that the "min" strategy works better than other initialization schemes. A reasonable conclusion from these observation is that lower temperature works better for knowledge distillation. Another key observation is that if we use an ensemble of deep neural networks, a richer knowledge is distilled in the student model. This can be observed from Figure 5.4.

## 5.3 Why does Experience Loss Work?

In the previous sections we have shown that Experience Loss significantly outperforms the baseline results obtained by a smaller network and is able to distill knowledge in the student network from multiple teacher models. In this section, we examine Experience Loss from a qualitative and experimental perspective to understand why it works and how it defines a scalable method for knowledge distillation.

We can think of neural networks as non-linear systems and the logits (or log probabilities) produced by a *trained* neural network to be a dirac-delta response that peaks on the correct label $y$ of the associated input pattern $x$. However, if we soften the normalized logits by a certain amount then the dirac-delta response is spread over some classes. This broader response gives the student more information about the output distribution of classes as a whole. With Experience Loss, we soften the logit response of the teachers at *different* temperatures and thus introduce diversity in the responses. For instance, a model $m_1$ might say that the input pattern belongs to class 3 with 98% confidence, while another model $m_2$ in the same ensemble may associate 3 with only 90% confidence. This diverse opinion amongst correct and incorrect classes transfers much more knowledge than just softening all logits at the same temperature.

Figure 5.4 shows the output heatmaps of logits obtained via Distillation and Experience Loss. The

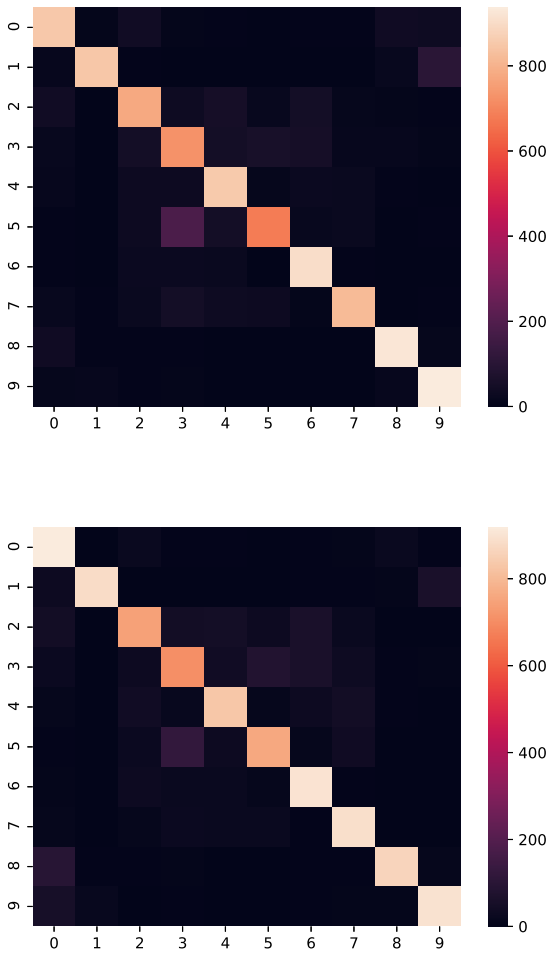


Figure 5.4: Comparison between output logits produced by Distillation and Experience Loss. Top: Distillation Loss
Bottom:Experience Loss(min strategy). [Best Viewed in Color]

lighter color on the diagonals indicate better performance. We observe that for almost all classes, Experience Loss has more correct predictions than Distillation Loss. To further validate our observations we trained Wide-Net (Section 5.1) and VGG-11 [34] as students using Distillation Loss and Experience Loss. Table 5.4 shows the result of our experiments.

| Model | Distillation Loss | Experience Loss | | |
|---|---|---|---|---|
| | | Mean | Max | Min |
| Wide-Net | 83.02 | 83.1 | 83.7 | 83.9 |
| VGG-11 | 83.35 | 83.46 | 83.40 | 83.87 |

Table 5.4: Performance of Distillation Loss wrt Experience Loss.

Both networks were trained for 50 epochs on the CIFAR-10 dataset. We observe that Experience Loss with min strategy outperforms Distillation Loss in both shallow and deep architectures. Experimentally, this lends strength to our earlier hypothesis that Experience Loss takes advantage of the diversity of an ensemble.

## 5.4 Learning with Partial Information

In Section 5.3 we hypothesized that the student network learns a much more complete information if the logits of the models in an ensemble are softened at non-uniform temperatures. However, the underlying assumption in the earlier experiments was that the student and teacher had access to the same training set $\mathcal{D}_{train}$. In the LUPI paradigm (Chapter 3) by Vapnik *et al.*we observed that the teachers could distill more knowledge if they had acess to some *privileged* information. In this section, we look at translating that paradigm to knowledge distillation in an ensemble.

### 5.4.1 Experimental Setup

We performed our experiments on the CIFAR-10 dataset. In order to generate a privileged dataset for the teachers, we removed all instances of the *frog* class from the student training set $\mathcal{D}_s$. Thus, from the student's viewpoint it has no idea of what a frog looks like. On the other hand, the teachers know of the entire input patterns and their associated class. Given this setup, we performed experiments with 3 different optimization strategies:

- Strategy 1: Student model trained on $\mathcal{D}_s$ with information from $y \in \mathcal{D}_s$.

- Strategy 2: Student model trained with one teacher and Distillation Loss.
- Strategy 3: Student model trained with two teachers and Experience Loss.

We chose Wide-Net as our student model of choice.

### 5.4.2 Results

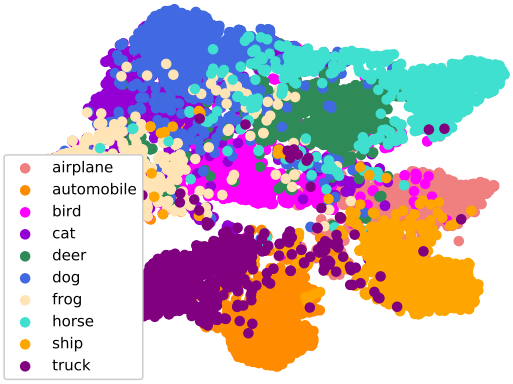


Figure 5.5: t-SNE Visulaziation of predicted classes from a student model.

A student that is trained with no information about a particular class will expectedly misclassify each of those samples. However, if we focus on how much the student misclassifies the information, then we can gain an understanding of how much knowledge is transferred to the student via distillation or experience loss. In our experiments, a student model trained via strategy 1 misclassified about 8.5% of the total images as belonging to class 5 ("dog"), while strategy 2 reduced it to about 8%. With strategy 3, the rate of misclassification to class 5 further dropped to 7.4%. Figure 5.5 shows the t-SNE visualization of the output distribution produced by the student with partial access to information.

## 5.5 Experiments with SVHN Dataset

In this section we discuss the performance of Experience Loss on the SVHN Dataset [28]. The SVHN dataset is another classification dataset that has data collected by Google's street view of different

| Teachers | Temperatures | Validation Accuracy |
|---|---|---|
| VGG-11 | [10] | 92.04 |
| VGG-11, Resnet-18 | [10, 20] | 95.59 |
| VGG-11, Resnet-18, Wide-Resnet | [10,1.5,20] | 95.73 |
| Baseline | - | 88.13 |

Table 5.5: Results on SVHN Dataset with different teacher models. In case of one teacher, Distillation Loss was used.

house numbers. Figure 5.6 shows some samples from the dataset.

Figure 5.6: Samples from the SVHN Dataset.

We trained the same teacher models as with Wide-Net, i.e. Wide-ResNet [36], VGG [34] and ResNet [12]. However, the student model we chose were simple 5 and 6 layer CNNs. All teacher models were trained for 100 epochs, while the student models were trained for 50 epochs. In order to measure the performance more accurately, we limited ourselves to 32x32 crops of single digits.

### 5.5.1 Results

We observed our insights from earlier experiments and used the "min" strategy when training a student model via experience loss. Table 5.5 shows the results of our experiments with different teacher models and a 6-layer CNN as a student. The higher accuracies can be explained by the fact that a single 32x32 image contains a single digit which is easily distinguishable. Table 5.6 shows the accuracies obtained by the teacher models on the SVHN dataset.

| Teacher Model | Validation Accuracy |
|---|---|
| Resnet-18 | 99.98 |
| VGG-11 | 99.86 |
| Wide Resnet | 97.53 |

Table 5.6: Best validation accuracy of teacher models with SVHN dataset. All teachers were trained for 100 epochs.

From Table 5.5 we observe a similar trend as with the CIFAR-10 dataset when the student model is trained with Experience Loss i.e. a model trained with Experience Loss performs better than the baseline and comparably with Distillation Loss. We also measured the class predictions by a 6 layer CNN when it is trained via two and three teachers. Figure 5.7 shows the class predictions as a bar-graph.

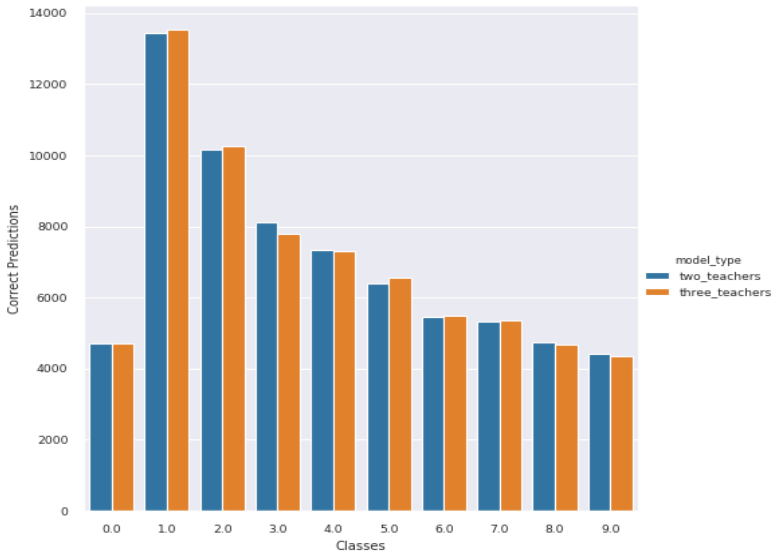


Figure 5.7: Class Predictions of a model trained with 2 and 3 teachers respectively.

| Ensemble Models | Total Ensemble Size (MB) | Compression Ratio($\phi$) | |
|---|---|---|---|
| | | Wide-Shallow | Wide-Deep |
| VGG-19, WideResnet | 218 | 357 | 12.82 |
| Resnet-18, WideResnet | 184 | 301 | 10.8 |
| VGG-11, WideResnet | 176 | 288 | 10.35 |
| VGG-11, VGG-19, WideResnet | 255 | 418 | 15 |
| VGG-11, VGG-19, Resnet-18 | 161 | 263 | 9.47 |

Table 5.7: Compression Results on Wide-Shallow and Wide-Deep Net on CIFAR-10 Dataset.

From the graph we can clearly see the contrast between the predictions of a model when trained with two and three teachers. A model trained with three teachers performs slightly better than a model trained with two teachers. We can explain this observation using our hypothesis in Section 5.3. Since an ensemble containing more teacher models introduces more diversity in the information transmitted to the student, measuring cross entropy between the different teacher and students leads to overall performance. On a shallow net architecture like a 6-layer CNN, we also note that a larger ensemble has a regularization effect that prevents it from overfitting.

## 5.6 Compression Results

In model compression a key question arises - how well does the model compress? For pruning based approaches, the answer is simply the size difference between the model obtained after pruning to the one before. On the other hand, there is no standard way to measure the effectiveness of compression in the knowledge distillation paradigm since the size of the student model is independent of the size of the teacher model(s).

If one observes the training procedure in a knowledge distillation paradigm then it becomes apparent that a student model, once trained under the influence of either a single teacher or an ensemble can be independently deployed on any platform. Thus, a way to quantify compression could be to compare the size of the student model to the *total* size of the ensemble. Mathematically, we can define the compression ratio as:

$$\phi = \frac{\sum_{i=0}^{N} \Omega(m_i)}{\Omega(s)} \tag{5.1}$$

In Equation 5.1, $\Omega(m_i)$ refers to the size of the $i^{th}$ model in the ensemble while $\Omega(s)$ refers to the size of the student model. Based on the formulation in Equation 5.1, we present the compression

ratios of the Wide-Shallow and Wide-Deep Net on the CIFAR-10 dataset in Table 5.7.

We used a PyTorch add-on called "torchsummary" to calculate the number of trainable parameters in a given model. Since we used weights in full precision, we multiplied the number of parameters by four and obtained the memory consumption. For Wide-Shallow Net (Section 5.1) the total memory was $0.61MiB$ while for the Wide-Deep Net (Section 5.2) the total memory was $17MiB$. Table 5.7 shows the total size of the ensemble along with the compression ratio of the Wide-Shallow and Wide-Deep Networks. We can see that our approach yields significantly smaller networks while increasing the baseline accuracy of these smaller networks to within tolerable range of the ensemble accuracy.

## 5.7 Chapter Summary

In this section we looked at experimental results with a varying number of teacher models in an ensemble. We introduced three initialization strategies for selecting the student temperature $T_z$. We then examined in some detail the relative effects of different teacher model architectures on our proposed student models. From the results it becomes clear that lower temperatures lead to better knowledge transfer in our proposed scheme. Additionally, we observe that ensemble of teacher models lead to better regularized student models. We conclude our work in the next chapter.

# Chapter 6

# Conclusion and Future Work

In this section, we conclude our work and provide avenues for future work and extensions to our original idea. We will begin by examining the insight gained from our experiments in Section 6.1 and then discuss the future work in Section 6.2

## 6.1 Conclusions

In this thesis we have examined the problem of model compression from first principles. We defined the need and theoretically motivated the problem of model compression. We then examined different approaches that have been proposed to solve the problem and then introduced our own novel contribution. This contribution enables *scalable* model compression, i.e. it gives a way to compress diverse knowledge from different sources into one small model, which is both energy and memory efficient. We also introduced two new architectures by drawing ideas from different related sources. These models are intended to be memory efficient as well as able to fit to different datasets and tasks. Finally, we looked at the reduction in memory consumption by using a student model trained under our scheme in place of the ensemble that trained the student model.

The core contribution of our work is in defining an end-to-end training algorithm to distill the knowledge from an ensemble of diverse deep neural networks at different softening temperatures. Additionally, we examine the underyling principles in memory efficient architecture design and propose ideas to make existing architectures less memory heavy.

Experimentally, we observe that the "min" scheme for selecting the student temperature consistently outperforms all other initialization schemes. A reasonable conclusion from the observation is that lower "temperature" leads to better knowledge transfer. We additionally observe two conflicting observations. On one hand, the accuracy of the student increases when the number of models in an ensemble are increased whereas on the other hand, the accuracy tends to plateau with three or more teachers. An explanation of these observations could be that as more models are added in an ensemble, the difference between the parameters of the ensemble and the student increases linearly and hence there must exist an upper bound on the number of models that can be fit into an ensemble for our given approach. We believe that this unknown upper bound on number of models is a potential drawback to our approach.

## 6.2 Future Work

We have defined a training algorithm, introduced two memory efficient networks, and developed a training strategy in our work. This leads to some exciting areas of research and extending our work in different directions. In this section, we examine some of the major areas of extending our work.

### 6.2.1 Automatic Temperature Selection

In our formulation, we have treated the temperature as a hyperparameter, i.e. a given constant that needs to manually adjusted. However, given the amount of datasets and the tasks that exist in deep learning, it is cumbersome to manually adjust and find the optimal hyperparameter for each and every data set. A good future work in this direction will be to identify techniques to infer the optimal softening temperature for a given dataset. For instance, the student temperature could be set to the harmonic mean of the teacher temperatures which could lead to much better convergence rates.

Another possibility to extend this would be incorporate the temperature prediction as a *layer* in the architecture. Similar to BatchNormalization [17] one can compute the optimal temperature for a given batch of data and keep a running track of this statistic. In this way, we can free the formulation from unnecessary manual tuning of an additional hyperparamter.

### 6.2.2 Extending Networks to Other Tasks

The two memory-efficient networks introduced in our thesis can be adapted to other tasks as well. For instance, the Wide-Deep Net achieves comparable performance to VGG-19 on the CIFAR-10 dataset. Thus, for tasks like object detection or instance segmentation the Wide-Deep Net can be used a *feature extractor* backbone. A direct advantage of this would be a significantly lighter detector or segmentation model. Additionally, one can formulate a scheme where the feature extractor backbone is trained from an ensemble, while the regressor focuses on the information from the ground truth label.

Another exciting direction of work is to focus on a more efficient DNN architecture that can compete with MobileNet [14] in terms of resource usage and yet have a higher accuracy due to being trained by our loss function.

### 6.2.3 Ensemble Methods

An area of possible extension lies in the direction of optimizing ensemble methods of machine learning for knowledge distillation. If we assume the ensemble to be solely comprised of neural networks then adaptive ensembles of teachers can be used to introduce even more diversity in the resulting logits. Currently, the ensemble methods are not fully explored in context of knowledge distillation and thus formulating innovative methods to capture the knowledge learned by the ensemble could be a valid direction of future research.

# Bibliography


[1] "An Updated Set of Basic Linear Algebra Subprograms (BLAS)". In: *ACM Trans. Math. Softw.* 28.2 (June 2002), pp. 135–151. ISSN: 0098-3500. DOI: 10.1145/567806.567807. URL: http://doi.acm.org/10.1145/567806.567807.

[2] E. Anderson et al. "LAPACK: A Portable Linear Algebra Library for High-performance Computers". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. Supercomputing '90. New York, New York, USA: IEEE Computer Society Press, 1990, pp. 2–11. ISBN: 0-89791-412-0. URL: http://dl.acm.org/citation.cfm?id=110382.110385.

[3] Cristian Buciluǎ, Rich Caruana, and Alexandru Niculescu-Mizil. "Model compression". In: *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '06* (2006), p. 535. ISSN: 0004-6361. DOI: 10.1145/1150402.1150464. arXiv: 1803.01118. URL: http://portal.acm.org/citation.cfm?doid=1150402.1150464.

[4] François Chollet. "Xception: Deep Learning with Depthwise Separable Convolutions". In: *CoRR* abs/1610.02357 (2016). arXiv: 1610.02357. URL: http://arxiv.org/abs/1610.02357.

[5] *CUDA Toolkit 4.2 CUBLAS Library*. NVIDIA Corporation, 2012.

[6] Yann Le Cun, John S Denker, and Sara a Solla. "Optimal Brain Damage". In: *Advances in Neural Information Processing Systems* 2.1 (1990), pp. 598–605. ISSN: 1098-6596. DOI: 10.1.1.32.7223. arXiv: arXiv:1011.1669v3.

[7] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR09*. 2009.

[8] Emily Denton et al. "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation". In: (2014), pp. 1–11. ISSN: 10495258. arXiv: 1404.0736. URL: http://arxiv.org/abs/1404.0736.

[9] Song Han, Huizi Mao, and William J. Dally. "Deep Compression - Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding". In: *Iclr* (2016), pp. 1–13. arXiv: 1510.00149. URL: http://arxiv.org/abs/1510.00149%7B%5C%%7D5Cnhttp://www.arxiv.org/pdf/1510.00149.pdf.

[10] Song Han et al. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". In: *SIGARCH Comput. Archit. News* 44.3 (June 2016), pp. 243–254. ISSN: 0163-5964. DOI: 10.1145/3007787.3001163. URL: http://doi.acm.org/10.1145/3007787.3001163.

[11] Babak Hassibi et al. "Optimal Brain Surgeon: extensions and performance comparisons". In: *Advances in Neural Information Processing Systems 6. Proceedings of the 1993 Conference* (1994), pp. 263–270.

[12] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: (2015). ISSN: 1664-1078. DOI: 10.3389/fpsyg.2013.00124. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[13] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. "Distilling the Knowledge in a Neural Network". In: (2015), pp. 1–9. ISSN: 0022-2488. DOI: 10.1063/1.4931082. arXiv: 1503.02531. URL: http://arxiv.org/abs/1503.02531.

[14] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[15] Forrest N. Iandola et al. "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡1MB model size". In: *CoRR* abs/1602.07360 (2016). arXiv: 1602.07360. URL: http://arxiv.org/abs/1602.07360.

[16] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009. ISBN: 630813-054US.

[17] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *CoRR* abs/1502.03167 (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[18] E. D. Karnin. "A Simple Procedure for Pruning Back-propagation Trained Neural Networks". In: *Trans. Neur. Netw.* 1.2 (June 1990), pp. 239–242. ISSN: 1045-9227. DOI: 10.1109/72.80236. URL: http://dx.doi.org/10.1109/72.80236.

[19] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980.

[20] Ranjay Krishna et al. "Visual Genome: Connecting Language and Vision Using Crowdsourced Dense Image Annotations". In: 2016. URL: https://arxiv.org/abs/1602.07332.

[21] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images*. Tech. rep. 2009. URL: https://www.cs.toronto.edu/%7B~%7Dkriz/learning-features-2009-TR.pdf.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances In Neural Information Processing Systems* (2012), pp. 1–9. ISSN: 10495258. DOI: http://dx.doi.org/10.1016/j.protcy.2014.09.007. arXiv: 1102.0183.

[23] Xu Lan, Xiatian Zhu, and Shaogang Gong. "Knowledge Distillation by On-the-Fly Native Ensemble". In: *CoRR* abs/1806.04606 (2018). arXiv: 1806.04606. URL: http://arxiv.org/abs/1806.04606.

[24] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *CoRR* abs/1405.0312 (2014). arXiv: 1405.0312. URL: http://arxiv.org/abs/1405.0312.

[25] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: To appear. 2016. URL: http://arxiv.org/abs/1512.02325.

[26] David Lopez-Paz et al. "Unifying distillation and privileged information". In: *International Conference on Learning Representations (ICLR 2016)*. 2016. URL: http://leon.bottou.org/papers/lopez-paz-2016.

[27] Asit K. Mishra and Debbie Marr. "Apprentice: Using Knowledge Distillation Techniques To Improve Low-Precision Network Accuracy". In: *CoRR* abs/1711.05852 (2017). arXiv: 1711.05852. URL: http://arxiv.org/abs/1711.05852.

[28] Yuval Netzer et al. *Reading Digits in Natural Images with Unsupervised Feature Learning*. Tech. rep. URL: http://ufldl.stanford.edu/housenumbers/.

[29] Adam Paszke et al. "Automatic differentiation in PyTorch". In: (2017).

[30] Antonio Polino, Razvan Pascanu, and Dan Alistarh. "Model compression via distillation and quantization". In: *International Conference on Learning Representations*. 2018. URL: https://openreview.net/forum?id=S1XolQbRW.

[31] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640.

[32] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *Advances in Neural Information Processing Systems 28*. Ed. by C. Cortes et al. Curran Associates, Inc., 2015, pp. 91–99. URL: http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-object-detection-with-region-proposal-networks.pdf.

[33] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2018.

[34] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: (2014), pp. 1–14. ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.005. arXiv: 1409.1556. URL: http://arxiv.org/abs/1409.1556.

[35] Vladimir Vapnik and Rauf Izmailov. “Learning Using Privileged Information: Similarity Control and Knowledge Transfer”. In: *Journal of Machine Learning Research* 16 (2015), pp. 2023–2049. URL: http://jmlr.org/papers/v16/vapnik15b.html.

[36] Sergey Zagoruyko and Nikos Komodakis. “Wide Residual Networks”. In: (2016). DOI: 10.5244/C.30.87. arXiv: 1605.07146. URL: http://arxiv.org/abs/1605.07146.

[37] Ying Zhang et al. “Deep Mutual Learning”. In: *CoRR* abs/1706.00384 (2017).