CODE: COMPILER-BASED NEURON-AWARE ENSEMBLE TRAINING

Ettore M. G. Trainiti [1] **Thanapon Noraset** [1] **David Demeter** [1] **Doug Downey** [1 2] **Simone Campanoni** [1]

ABSTRACT

Deep Neural Networks (DNNs) are redefining the state-of-the-art performance in a variety of tasks like speech recognition and image classification. These impressive results are often enabled by ensembling many DNNs together. Surprisingly, ensembling is often done by training several DNN instances from scratch and combining them. This paper shows that there is significant redundancy in today's way of ensembling. The novelty we propose is CODE, a compiler approach designed to automatically generate DNN ensembles while avoiding unnecessary retraining among its DNNs. For this purpose, CODE introduces neuron-level analyses and transformations aimed at identifying and removing redundant computation from the networks that compose the ensemble. Removing redundancy enables CODE to train large DNN ensembles in a fraction of the time and memory footprint needed by current techniques. These savings can be leveraged by CODE to increase the output quality of its ensembles.

1 INTRODUCTION

Deep Neural Networks (DNNs) are redefining the state-of-the-art performance on a growing number of tasks in many different domains. For example, in speech recognition, encoder-decoder DNNs have set new benchmarks in performance (Chiu et al., 2017). Likewise, the leaderboards of the ImageNet image classification challenge are dominated by DNN approaches such as residual convolutional neural networks (He et al., 2016; Wang et al., 2017). These impressive results have enabled many new applications and are at the heart of products like Apple Siri and Tesla AutoPilot.

DNNs achieve high-quality results after extensive training, which takes a significant amount of time. The importance of this problem has been highlighted by Facebook VP Jérôme Pesenti, who stated that rapidly increasing training time is a major problem at Facebook (Johnson, 2019).

There are two main reasons why DNN training takes so long. First, training a single DNN requires tuning a massive number of parameters. These parameters define the enormous solution space that must be searched in order to find a setting that results in strong DNN output quality. Optimization techniques that are variants of stochastic gradient descent are effective at finding high quality DNNs, but require many iterations over large datasets. Today, several powerful hardware accelerators like GPUs or TPUs are necessary to explore this space fast enough to keep the training time down to acceptable levels. The second reason that results in increased DNN training time is a technique known as ensembling: often, the best output-quality is achieved by training multiple independent DNN instances and combining them. This technique is common practice in the machine learning domain (Deng & Platt, 2014; Hansen & Salamon, 1990; Sharkey, 2012). This paper aims at reducing training time resulting from homogeneous DNN ensembling.

Ensembling is leveraged in a variety of domains (e.g., image classification, language modeling) in which DNNs are employed (He et al., 2016; Hu et al., 2018; Jozefowicz et al., 2016; Liu et al., 2019; Wan et al., 2013; Yang et al., 2019). In all of these domains the approach used to ensemble N DNNs is the following. After randomly setting the parameters of all DNN instances, each DNN instance has its parameters tuned independently from any other instance. Once training is completed, the DNN designer manually ensembles all these trained DNNs. To do so, the designer introduces a dispatching layer and a collecting layer. The former dispatches a given input to every DNN that is participating in the ensemble. The latter collects the outputs of these DNNs and aggregates them using a criterion chosen by the designer. The output of the DNNs ensemble is the result of this aggregation.

We believe there is an important opportunity to improve the current DNNs ensembling approach. The opportunity we found lies in a redundancy we observed in the training of the networks that together compose the ensemble. We observed this redundancy exists because there is a common sub-network between these DNN instances: this sub-network

[1]Department of Computer Science, Northwestern University, Evanston, IL, USA
[2]Allen Institute for AI, Seattle, WA, USA. Correspondence to: Ettore M. G. Trainiti <ettrai@u.northwestern.edu>.

Proceedings of the 4^{th} MLSys Conference, San Jose, CA, USA, 2021.

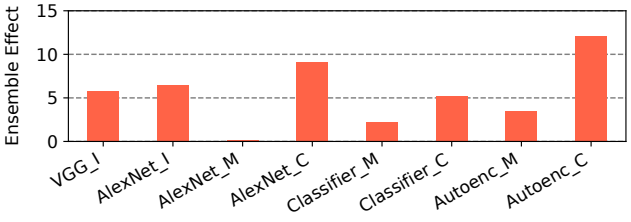

Figure 1. An ensemble of DNNs performs better than a single DNN. The ensemble effect is the output quality difference between an ensemble and the best single DNN that participates in it. The output quality of the benchmarks are reported in Table 1.

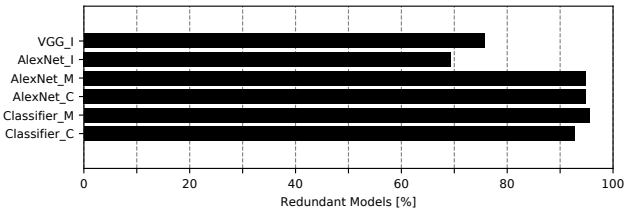

Figure 2. The DNNs that compose an ensemble often generate the same outcomes, making their contribution redundant.

represents a semantically-equivalent computation. Training this sub-network from scratch for all DNN instances that compose the ensemble is unnecessary. This sub-network spans across layers of the neural network; hence, detecting it requires analyses that reach the fine granularity of a neuron. This requirement suggests the need for an automatic tool. In this paper we introduce this tool: CODE, the Compiler Of Deep Ensembles.

CODE is a compiler that automatically trains and ensembles DNNs while significantly reducing the ensemble's training time by avoiding retraining the common sub-network. Similar to conventional compilers that involve Code Analysis and Transformation (CAT) to recognize and remove redundant computation performed by the instructions of a program (Aho et al., 1986), CODE introduces Neuron Analysis and Transformation (NAT) to recognize and remove redundant computation performed by the trained neurons of a DNN. The neuron-level redundancy elimination performed by the NATs introduced in this paper is what allows CODE to train an ensemble of DNNs much faster than current approaches.

We tested CODE on 8 benchmarks: CODE significantly reduces the ensemble training time of all of the benchmarks considered. In more detail, CODE trains homogenous ensembles of DNNs that reach the same output quality of today's ensembles by using on average only 43.51% of the original training time. Exploiting the training time savings achieved by CODE can result in up to 8.3% additional output quality while not exceeding the original training time. Furthermore, CODE generates ensembles having on average only 52.38% of the original memory footprint. When free from training time constraints, CODE reaches higher output quality than current ensembling approaches.

The paper we present makes the following contributions:

(i) We show the redundancy that exists in today's approaches to train homogeneous ensembles of DNNs

(ii) We introduce Neuron Analyses and Transformations to automatically detect and remove redundant training

(iii) We present the first neuron-aware compiler capable of automatically ensembling DNNs while significantly reducing unnecessary neuron retraining

(iv) We demonstrate the potential of CODE that exists even when relying only on commodity hardware.

2 Opportunity

A DNN ensemble generates higher-quality outputs compared to a single DNN instance (Jozefowicz et al., 2016). We validated this effect on the DNNs considered in this paper. To measure this effect, we compared the output quality of an ensemble of independently trained DNNs with the one obtained by a single DNN. Figure 1 shows their difference, also known as *ensemble effect*, for the benchmarks described in Section 4 and reported in Table 1. The benchmark naming convention we used is `Network_Dataset` (e.g., VGG trained on the ImageNet dataset is labeled `VGG_I`).
All benchmarks but `AlexNet_M` have important ensemble effects. `AlexNet_M` shows a small ensemble effect (+0.13%) because the quality of the single DNN instance is already high (99.21%), leaving almost no room for improvements. This ensemble effect results in a final ensemble output accuracy of 99.34%.

The ensemble effect largely results from *ensemble heterogeneity*: different DNN instances within the ensemble are likely to reside in different local minima of the parameter solution space. These different optima are a consequence of the different initial states that each DNN started training from. An ensemble of DNNs takes advantage of this local minima heterogeneity.

Our **observation** is that the heterogeneity of the DNNs local optima is also shown through the DNNs output differences. While these differences are fundamental to generate the ensemble effect (Figure 1), we observe that output differences exist only for a small fraction of the network instances within a DNN ensemble. To examine this characteristic, we considered the output outcomes originating from DNNs used for classification tasks. Figure 2 shows the fraction

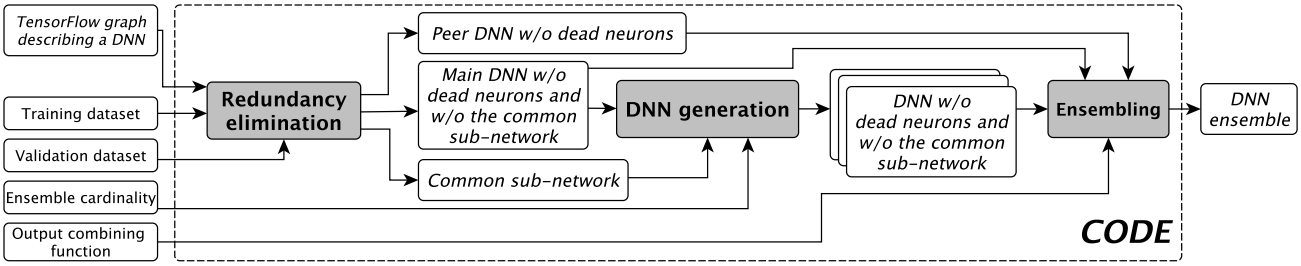

Figure 3. The CODE methodology. Highlighted elements are the three phases of CODE.

of independently trained DNN instances that generate an output that was already generated by at least another DNN instance of the ensemble. These fractions suggest the existence of redundancy between the networks participating in the ensembles. To compute the values shown in Figure 2, we applied the following formula:

$$R = \frac{\sum_{i=1}^{I} (N - U_i)}{N \times I}$$

Where R is the redundancy, N is the number of networks that compose the ensemble (shown in Table 1), I is the number of the inputs, and for a given input i the term U_i is the number of unique outputs produced by the DNNs part of the ensemble.

Our **research hypothesis** is that there is a significant amount of redundancy across DNN instances that participate in a DNN ensemble. In other words, some intrinsic aspects of the inputs are equally learned by all the DNN instances within the ensemble. We call these equally-learned aspects the *common sub-network* of an ensemble. We believe that re-learning the common sub-network for all DNN instances that compose an ensemble is not strictly needed. This unnecessary redundancy can be removed.

Our empirical evaluation on homogeneous ensembles in Section 4 strongly suggests that our research hypothesis is valid. This hypothesis gave us a new **opportunity**: to automatically detect the common sub-network by comparing N DNN instances. Once identified, the common sub-network can be extracted and linked to new DNNs ahead of training such that it will be part of their initialization. Training these additional DNN instances will only require tuning parameters that do not belong to the common sub-network of the ensemble. In particular, the new DNNs that will be part of the ensemble will only "learn" aspects of the inputs that contribute to the ensemble effect. This optimization is automatically implemented and delivered to the end user by CODE.

3 CODE METHODOLOGY

CODE is capable of automatically ensembling DNNs while limiting unnecessary parameter retraining thanks to transformations specifically designed to analyze, identify, and remove inter-network neuron redundancy. An overview of CODE's approach is shown in Figure 3, reported above.

The savings obtained by CODE (both in terms of training time and memory used by a DNN ensemble) come from removing two sources of redundancy we have identified. The first source of redundancy is a consequence of the fact that DNNs are often over-sized. This source of redundancy can be detected by analyzing the contribution of a neuron to the DNN outputs. A neuron can be safely removed if, for all of its inputs, it does not contribute to the final outputs of the DNN. We call these neurons *dead neurons*. The second and main source of redundancy comes from the existence of common sub-networks. Common sub-networks are collections of connected neurons that behave in a semantically-equivalent way across different networks. After the detection and confinement of this novel source of redundancy, the training of new DNNs to be added to the ensemble will only cover parameters that are not part of the extracted common sub-network. What follows is the description of the three phases carried by CODE: *Redundancy Elimination*, *DNN generation*, and *Ensembling*.

3.1 Phase 1: Redundancy Elimination

This phase starts with the training of two DNNs that follow the DNN architecture given as input to CODE. These two training sessions, including their parameters initialization, are done independently. We call these trained models *main* and *peer* DNNs.

We chose to train conventionally only the main and peer DNNs because of two reasons. The first is that when more DNNs are trained conventionally, CODE has less room to reduce ensemble training time. The training time saved by CODE comes from unconventionally training the other DNNs that participate in the ensemble. The second reason is that analyzing only the main and the peer DNNs led us to already obtain important training time savings as Section 4

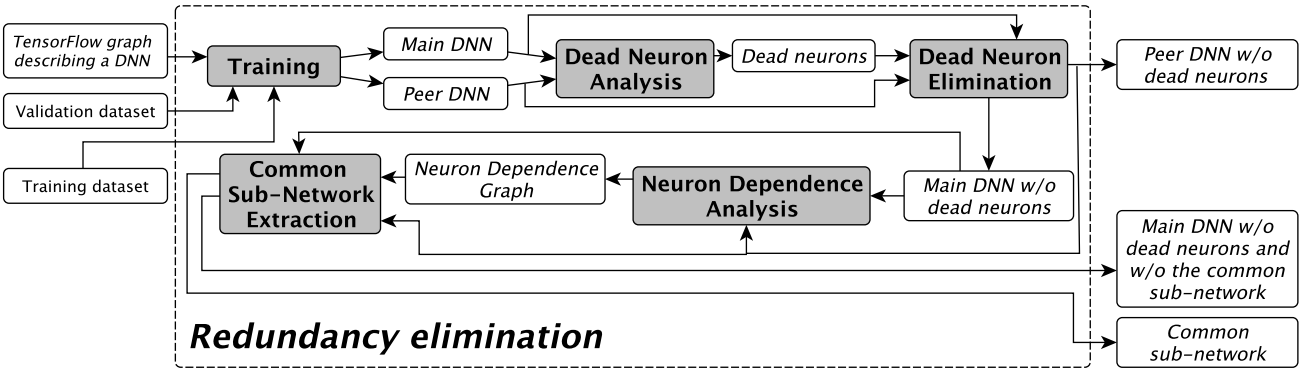

Figure 4. The redundancy elimination phase of CODE.

demonstrates empirically. It is anyways possible for CODE to analyze more than two conventionally-trained DNNs.

Once the main and peer DNNs are trained, CODE can start the identification of the first source of redundancy thanks to its Dead Neuron Analysis (DNA). DNA detects all the dead neurons of a DNN. We consider a neuron to be dead when its activations do not influence the outputs of the DNN it belongs to. To this end, the DNA checks if the activations of a neuron are zero for all inputs considered. Neurons that meet this condition are considered dead. The DNA included in CODE conservatively generates the list of dead neurons from the intersection of neurons that are dead in all DNNs given as input. This differs from the conventional approach of deleting dead neurons from a single DNN after its training. In more detail, we chose to delete *only* the neurons that are dead for *all* DNNs because a neuron can be dead in a network, but alive in another. This subtle difference can contribute to the ensemble DNNs' heterogeneity which translates into better ensemble output quality. To make sure this assumption is not too conservative, we tested it. We found our assumption to be true in the benchmarks we considered: removing dead neurons that are alive in one network but dead in the other (i.e., the conventional approach) significantly reduced the ensemble output quality.
The list of neurons generated by the DNA is then given to the Dead Neuron Elimination (DNE) component. This component removes those neurons from a given network, in our case, it removes them from the main and peer models by properly modifying and reshaping their parameter tensors. This concludes the removal of the first source of redundancy identified by CODE.

The second source of redundancy to be removed by CODE is the one given by the existence of common sub-networks across homogeneous DNNs. To start the identification of neurons that will become part of the common sub-network, CODE performs what we call the Neuron Dependence Analysis (NDA). The idea that inspired our NDA is simple: connected neurons that strongly influence each other need to be kept together. In machine learning this concept is called "firing together" and it is used in Hebbian learning (Hebb, 1949; Hecht-Nielsen, 1992) for hopfield networks. NDA leverages the activations of each pair of connected neurons in a DNN to understand whether they should be connected in the Neuron Dependence Graph (NDG). To do this, NDA counts the fraction of inputs that makes both neurons "fire" together. For this test, we encoded a custom firing condition for each different neuron activation function. Connected neurons that often fire together are linked by an edge in the NDG. An edge between nodes of the NDG represents the constraint that connected neurons cannot be separated. In other words, either they both are selected to be part of the common sub-network or neither one is. The output of the NDA is the NDG where all identified neurons dependencies of a DNN are stored.
Once the NDGs of the main and peer network have been obtained, CODE can perform its most crucial step: the Common Sub-Network Extraction (CSNE). The CSNE identifies and removes the set of semantically-equivalent neurons between the main and the peer DNNs while satisfying the constraints specified in the NDGs. The CSNE extracts these neurons from the main DNN by reshaping its tensors and placing them in what is going to be the Common Sub-Network (CSN). We define semantically-equivalence as follows. A neuron n_m of the main DNN is semantically-equivalent to a neuron n_p of the peer DNN if and only if all the following conditions are met:

(i) n_m and n_p belong to the same layer in their correspondent DNNs

(ii) n_m and n_p fire together often enough (e.g., more than 80% of the time, across the inputs considered)

(iii) there is a high correlation (e.g., more than 0.8) between the sequence of activations of n_m and the sequence of activations of n_p

(iv) all the predecessor neurons that n_m depends on are part of the common sub-network.

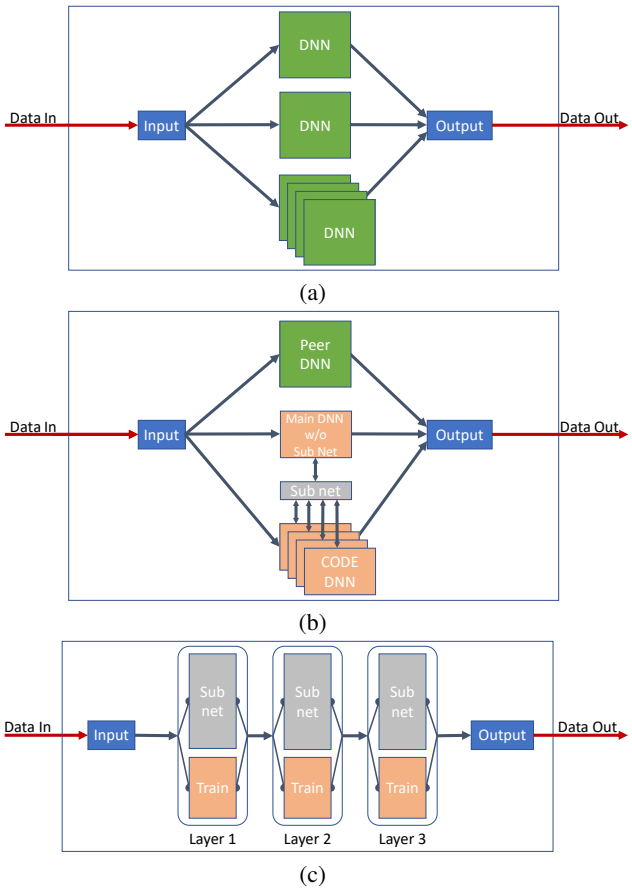

Figure 5. CODE ensembles are structurally different from vanilla ensembles.
(a) Ensemble made of independently trained networks
(b) Ensemble generated by CODE
(c) Structure of a single CODE DNN instance

Condition (i) makes sure that n_m and n_p belong to a compatible location (e.g., layer1) in their correspondent DNNs. Conditions (ii) and (iii) check that n_m and n_p behave similarly for most inputs considered. Condition (iii) relies on the conventional correlation formula (Rice, 2006) reported here for convenience:

$$Correlation_{n_m,n_p} = \frac{\text{cov}(M,P)}{\sigma_M \sigma_P}$$

where M and P are the vectors of the outputs of n_m and n_p. Finally, condition (iv) guarantees that the dependencies of neuron n_m, specified in the NDG of the DNN n_m belongs to, are satisfied.

The algorithm used by CSNE flags semantically-equivalent neurons until no neurons satisfy the conditions specified above. When convergence is reached, the set of neurons to be added to the common sub-network is removed from the main model via functional preserving transformations. The removed neurons are then packed into a collection of tensors that represent the extracted common sub-network. This common sub-network that will be used by CODE during its next phase called *DNN generation*. In Appendix A we show a direct comparison between the impact of DNE and CSNE. For completeness, we also tried to randomly select neurons to be added to the common sub-network: this led to obtain critical losses in terms of the output quality of the final ensembles. In Appendix B we share more about our random neuron selection trials.

3.2 Phase 2: DNN Generation

Having already trained the main and peer models, the DNN generation phase of CODE deals with the training of the remaining $(N-2)$ DNNs that will be part of the ensemble. The only parameters of these DNNs that need to be trained are the ones not included in the common sub-network extracted by the redundancy elimination phase. These trainable parameters are randomly initialized before training. In addition, the necessary common sub-network is linked to each DNN before its training. The result of this phase is $(N-2)$ trained CODE DNNs. One example of CODE DNN is shown in Figure 5c. The parameters that will be trained in a CODE DNN are labeled as "Train".

3.3 Phase 3: Ensembling

The Ensembler of CODE combines N DNNs to output the final DNN ensemble. In more detail, the Ensembler links the common sub-network to the main DNN and to the $(N-2)$ CODE DNNs. It then adds these networks and the peer DNN to the collection of DNNs that will be ensembled. The Ensembler extends this collection by introducing a component to distribute the input to all N DNNs and to collect all the DNN outputs. The DNNs outputs are combined as specified by the ensembling criterion provided to CODE by the user. Figure 5b shows the ensemble structure that the Ensembler outputs as a TensorFlow graph. Thus, the off-the-shelf TensorFlow stack can be used to perform inference on the DNN ensemble.

3.4 Relationship with Compilers

Our approach takes inspiration from techniques used in the Computer Systems domain. Within the Compiler community, these techniques are called Code Analysis and Transformations (CATs). Given the nature of DNNs workloads and their finer grain units, we decided to call ours Neuron Analysis and Transformations (NATs). Specifically, in this work we have defined and implemented four NATs: Dead Neuron Analysis (DNA), Dead Neuron Elimination (DNE), Neuron Dependence Analysis (NDA), Common Sub-Network Extraction (CSNE). DNA and DNE are similar in spirit to the dead code elimination CAT, NDA and CSNE are inspired by common sub-expression elimination performed by conventional compilers (Aho et al., 1986).

3.5 Implementation

CODE lives in ∼30,000 lines of code. We wrote CODE using a combination of Python and C++ code. Python has been used to interact with the TensorFlow stack. C++ has been used to implement the core of CODE, including all NATs described in Section 3.1. The time spent inside NATs is negligible (less than 0.05% of the total training time) thanks to a careful parallelization and vectorization of their code using OpenMP 4.5.

4 Empirical Evaluation

We evaluated CODE on several benchmarks and we compared its results with a baseline of homogeneous ensembles of independently trained DNNs. All experiments leveraged TensorFlow r1.12 on a single Nvidia GTX 1080Ti GPU.

4.1 Benchmarks

The naming convention we used for the benchmaks we evaluated is `Network_Dataset`. For example, AlexNet trained on the MNIST dataset is called `AlexNet_M`.

The set of networks we tested CODE on include the following DNNs:

Autoencoder: an unsupervised learning network. It is made of five fully connected hidden layers: encode1, encode2, code, decode1, decode2. Layers encode1 and decode2 have the same neuron cardinality, and so do layers encode2 and decode1. Autoencoder's task is to generate a noiseless image starting from a noisy input image. Its output quality over a dataset is measured in terms of mean squared error (MSE) between the original images and the output images. Our network code is an adapted version of (Kazemi) where convolutional layers have been replaced with fully connected layers, and image noise has been obtained by applying dropout (Srivastava et al., 2014) to the input layer.

Classifier: a network meant to learn how to classify an input over a number of predefined classes. Its structure consists of four fully connected hidden layers with different neuron cardinalities. Its output quality over a dataset is measured in terms of accuracy, i.e. correct predictions over total predictions. We adapted our code from the DNN shown in (Sopyla).

AlexNet (Krizhevsky et al., 2012): a network designed for image classification. This network comes in two variants: four hidden layers(Krizhevsky, a) and eight hidden layers (Krizhevsky, b). We adapted our code from the version distributed by Google (Google, a). Unlike the other two networks, AlexNet makes use of convolutional layers. These particular layers take advantage of the spatial location of the input fed to them. AlexNet output quality is measured in terms of accuracy.

VGG (Simonyan & Zisserman, 2014): a deeper network designed for image classification. This network comes in six different configurations: we used configuration D. We adapted our code from the version distributed by Google (Google, b). Like AlexNet, this network leverages convolutional layers. A peculiarity of this network is the use of stacked convolutional layers. VGG output quality is measured in terms of accuracy.

The datasets we used for our tests include:

MNIST (LeCun et al.): a dataset composed of 70000 entries containing handwritten digits. Each entry is pair of a 28x28 pixels gray-scale image and its corresponding digit label.

CIFAR10 (Krizhevsky et al.): a dataset of 60000 images of objects categorized in 10 different classes. Each 32x32 pixels color image is labeled according to the class it belongs to, e.g. "airplane", "automobile", "dog", "frog".

ImageNet (Russakovsky et al., 2015): one of the state-of-the-art datasets for image classification and object detection. The variant we used (ILSVRC, a) contains over 1.2 million entries of color images distributed over 1000 classes. These images have different sizes: we resized them to have a minimum length or height of 256 pixels and took the central 256x256 pixels crop, as described in (Krizhevsky et al., 2012).

For each dataset, with the exception of ImageNet, we did not perform any data preprocessing other than normalizing the input features of each image (from pixel with values [0,255] to [0,1]).

When training AlexNet on ImageNet, we computed the mean image over the whole dataset and then subtracted it from every image given as input to the network. When training VGG on Imagenet, we computed the mean RGB pixel over the whole dataset and then subtracted it from every pixel of any image given as input to the network.
During training on ImageNet, we performed data augmentation by providing the network with one random 224x224 pixels image crop taken from the initial 256x256 pixels central crop.

4.2 Ensembling criteria

We considered multiple ensembling criteria, including majority voting, averaging, products of probability distributions, and max. We chose the ones that gave the best output quality for ensembles of independently trained DNNs.

When **accuracy** is the metric used to measure the output quality of a model, we apply the following ensembling criterion. For an ensemble of N models that outputs K class probabilities per input, the ensemble output probabilities for

Table 1. Reported below are training time and output quality of baseline ensembles of independently trained DNNs to reach diminishing returns on our test platform. We also report the time needed for CODE to reach or beat the output quality achieved by the baseline.

Benchmark	Network	Dataset	Metric	Single Model	Ensemble	Ensemble Training Time	Cardinality	Approach
VGG_I	VGG-16	ImageNet	Top-1 Accuracy	57.82%	63.66%	40 d 09 h 56 m	11	Baseline
					63.91%	29 d 04 h 00 m	8	CODE
AlexNet_I	AlexNet	ImageNet	Top-1 Accuracy	46.12%	52.63%	5 d 07 h 44 m	12	Baseline
					52.70%	3 d 11 h 13 m	6	CODE
AlexNet_M	AlexNet	MNIST	Top-1 Accuracy	99.21%	99.34%	1 h 22 m	20	Baseline
					99.34%	1 h 06 m	19	CODE
AlexNet_C	AlexNet	CIFAR10	Top-1 Accuracy	69.05%	78.21%	7 h 30 m	45	Baseline
					78.22%	3 h 36 m	48	CODE
Classifier_M	Classifier	MNIST	Top-1 Accuracy	95.76%	98.03%	15 m	25	Baseline
					98.03%	5 m	23	CODE
Classifier_C	Classifier	CIFAR10	Top-1 Accuracy	48.90%	54.11%	36 m	31	Baseline
					54.12%	4 m	33	CODE
Autoenc_M	Autoencoder	MNIST	MSE	9.87	6.41	11 m	21	Baseline
					6.40	4 m	12	CODE
Autoenc_C	Autoencoder	CIFAR10	MSE	72.30	60.17	48 m	40	Baseline
					59.90	26 m	42	CODE

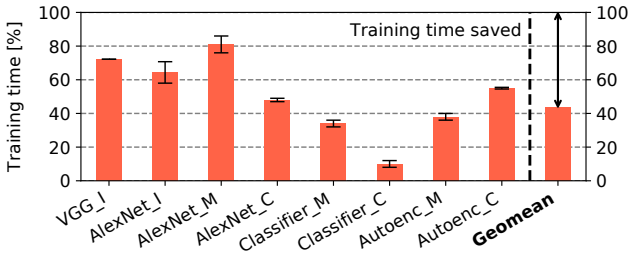

Figure 6. Training time needed by CODE to achieve the same output quality of the baseline of Table 1.

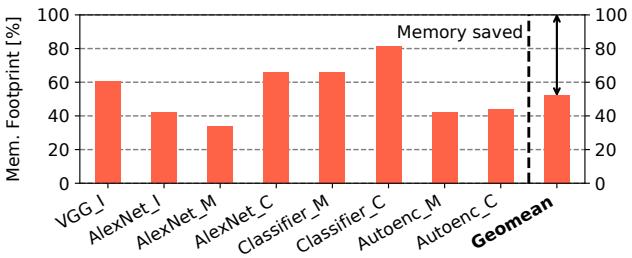

Figure 7. CODE Ensembles memory footprint with respect to baseline for the entries of Table 1.

the i^{th} input are computed as:

$$P_{ik} = \prod_{j=1}^{N} O_{ijk}$$

Where P_{ik} is the predicted probability of input i to belong to class k, O_{ijk} is the output probability for class k of the j^{th} model in the ensemble for the i^{th} input. The predicted label L_i of a given input is computed as

$$L_i = max\left\{P_{i1}, \ldots, P_{iK}\right\}$$

The accuracy metric is computed as the number of correctly predicted labels over the total number of inputs. The higher the accuracy, the better.

When cumulative **MSE** is the metric used to measure the output quality of model, we apply the following ensembling criterion. For an ensemble of N models where K is the number of dimensions of each input, the ensemble output value for the k^{th} dimension of the i^{th} input is computed as:

$$V_{ik} = \frac{\sum_{j=1}^{N} O_{ijk}}{N}$$

Where O_{ijk} is the output value for dimension k of the j^{th} model in the ensemble for the i^{th} input. The MSE is then computed between the ensemble output V_i and the expected output E_i. The sum of all the MSEs gives us the cumulative MSE metric. The lower the MSE, the better.

4.3 DNN Training

We leveraged the off-the-shelf Tensorflow software stack to train every DNN on a single GPU. Parallel training of each single DNN is anyways possible by leveraging frameworks such as Horovod (Sergeev & Del Balso, 2018). Learning rate tuning has been done for the baseline runs, and the same learning rate has been used for CODE runs. Weight decay has not been used during training and learning rate decay has only been used for `AlexNet_I` and `VGG_I` benchmarks. Early stopping has been leveraged to halt the training session when no output quality improvement on the validation set has been observed for 5 consecutive training epochs.

The measurements we report have been obtained by running each experiment multiple times. `AlexNet_I` has been run 3 times and `VGG_I` has been run only once due to the extensive amount of time it required to complete a single

run on our machine. All other benchmarks shown in Table 1 have been run for 30 times. We reported the output quality median of baseline and CODE ensembles, together with their training times, in Table 1.

4.4 Results

Training time savings Training a DNN ensemble requires the choice of its *cardinality*, i.e. how many models will be part of the ensemble. For each entry in Table 1, we reported the baseline ensemble cardinality $N_{Baseline}$ that led to diminishing returns in terms of output quality scored by each of our benchmarks. Baseline ensembles are made of independently trained networks.
CODE ensembles have been trained by following the approach described in Section 3. For each benchmark, we measured the total training time needed by CODE to obtain at least the same output quality of the baseline ensemble and reported its cardinality N_{CODE}. The results of these comparisons are reported in Table 1 and Figure 6. Thanks to its NATs, CODE obtained the output quality of the baseline ensembles in a fraction of the time. On average, CODE reaches the same or better baseline ensemble output quality using only 43.51% of the baseline training time. The combined advantages of redundancy reduction and avoiding the retraining of the common sub-network are the keys that led CODE to achieve important training time savings. It is worth noting that, on our machine, CODE NATs accounted for less than 0.05% of the total ensemble training time.

For completeness, we computed the total memory footprint of the CODE ensembles reported in Table 1. To do so, we used the following formula:

$$M = \frac{TP_{DNN} + (N_{CODE} - 1) \cdot TP_{CODE_{DNN}} + CSN}{N_{Baseline} \cdot TP_{DNN}}$$

Where M is the memory footprint of the CODE ensemble with respect to the baseline ensemble, TP is the number of trainable parameters, CSN is the number of non-trainable parameters stored in the common sub-network, and N_{CODE} and $N_{Baseline}$ are ensemble cardinalities. These results are shown in Figure 7. On average, CODE ensembles have 52.38% of the baseline memory footprint. These memory savings result from the sharing of the common sub-network enforced by CODE.

Higher output quality The training time savings we just described can be exploited by CODE to achieve higher output quality. While not exceeding the training times of the baseline ensembles specified in Table 1, CODE can invest those savings by training more DNNs to add to its ensembles. In Table 2 we show the extra output quality that CODE obtained when given the same training time budget as the baseline. CODE increases the output quality of most benchmarks (up to 8.3%) without exceeding the training time used by the baseline.
For completeness, we also measured and reported the extra output quality gained when no training time constraints are given to both the baseline and CODE, i.e. when any number of models can be added to the ensemble. We observed that the baseline does not meaningfully increase its output quality even when its training time budget is left unbounded. On the other hand, CODE obtains higher output quality out of its trained ensembles. More on this phenomenon in Appendix C.

5 Related Work

We show the relevance of ensembling DNNs and present the spectrum of community investments. We divide prior work related to improving DNNs into multiple categories: fast ensembling, optimizing training time, optimizing inference time, software stacks.

5.1 Ensembling Deep Neural Networks

Combining the outputs of multiple DNNs is a common technique used to perform better on a given metric (e.g. accuracy, MSE) for a given task (e.g. classification, object detection). Ensembling can be performed across networks (e.g. Inception-V4 (Szegedy et al., 2017), GoogLeNet (Szegedy et al., 2015)) and/or within networks (Shazeer et al., 2017). The outcomes of the last five (ILSVRC, b;c;d;e;f) ImageNet Large Scale Visual Recognition Challenges (Russakovsky et al., 2015) are a strong proof of this trend: ensembling models proved to be the winning strategy in terms of task performance. In unsupervised language modeling, ensembles of multiple models often outperform single models by large margins (Jozefowicz et al., 2016). Another example of the importance of ensembling is shown on the MNIST classification task. (LeCun et al.) reports the best accuracy obtained by a multitude of different networks. The entry that set the best tracked result is an ensemble of 35 Convolutional Neural Networks. To the best of our knowledge, ours is the first work to propose and evaluate methods aimed specifically at optimizing the training of ensembles for general DNNs while preserving output quality.

5.2 Fast Ensembling

A standard baseline for ensembling DNNs is to aggregate a set of independently trained deep neural networks. To reduce ensemble training time, existing fast ensembling techniques rely on two main ideas: transfer learning and network architecture sharing.

Transfer learning based approaches include Knowledge Distillation (Hinton et al., 2015) and Snapshot Ensembles (Huang et al., 2017). Knowledge Distillation (Hinton et al., 2015) method trains a large *generalist* network combined

Table 2. Reported below is the ensemble output quality when the same training time budget is given to baseline and CODE. We also show the output quality obtained by the baseline and CODE when no training time budget constraint is given.

Benchmark	Network	Dataset	Metric	Training Time Budget	Ensemble	Approach
VGG_I	VGG-16	ImageNet	Top-1 Accuracy	40 d 9 h 56 m	63.66%	Baseline
					64.02%	CODE
				Unbounded	63.99%	Baseline
					64.68%	CODE
AlexNet_I	AlexNet	ImageNet	Top-1 Accuracy	5 d 7 h 44 m	52.63%	Baseline
					53.24%	CODE
				Unbounded	53.16%	Baseline
					53.87%	CODE
AlexNet_M	AlexNet	MNIST	Top-1 Accuracy	1 h 22 m	99.34%	Baseline
					99.54%	CODE
				Unbounded	99.37%	Baseline
					99.836%	CODE
AlexNet_C	AlexNet	CIFAR10	Top-1 Accuracy	7 h 30 m	78.21%	Baseline
					86.21%	CODE
				Unbounded	79.21%	Baseline
					93.21%	CODE
Classifier_M	Classifier	MNIST	Top-1 Accuracy	15 m	98.03%	Baseline
					98.23%	CODE
				Unbounded	98.12%	Baseline
					98.83%	CODE
Classifier_C	Classifier	CIFAR10	Top-1 Accuracy	36 m	54.11%	Baseline
					62.41%	CODE
				Unbounded	55.11%	Baseline
					66.41%	CODE
Autoenc_M	Autoencoder	MNIST	MSE	11 m	6.41	Baseline
					6.21	CODE
				Unbounded	6.17	Baseline
					6.01	CODE
Autoenc_C	Autoencoder	CIFAR10	MSE	48 m	60.17	Baseline
					58.07	CODE
				Unbounded	59.73	Baseline
					54.87	CODE

with multiple *specialist* networks trained on specific classes or categories of a dataset. The Snapshot Ensembles (Huang et al., 2017) method averages multiple local minima obtained during the training of a single network.

Network architecture sharing based techniques include TreeNets (Lee et al., 2015) and MotherNets (Wasay et al., 2020). The idea behind TreeNets (Lee et al., 2015) is to train a single network that branches into multiple sub-networks. This allows all the learners to partially share few initial layers of the networks. Along this line of thinking, MotherNets(Wasay et al., 2020) targets structural similarity to achieve faster training times. At its core, MotherNets first trains a “maximal common sub-network" among all the learners and then transforms it back into the original models' structures through functional preserving transformations. All these models are then further trained until convergence is reached.

All the works mentioned above do not preserve the output quality of the baseline ensembles of independently trained networks when training faster than baseline, while CODE manages to do so.

CODE, relies on both transfer learning and network architecture sharing by focusing on functional (rather than structural) similarity. This enabled CODE to preserve the baseline accuracy while delivering training time savings, a substantial improvement over previous approaches. The core

idea of CODE is to avoid unnecessary redundant training of neurons that exist in ensembles of neural networks. CODE achieves this goal by identifying and extracting semantically-equivalent neurons from networks that have the same architecture.

5.3 Optimizing Training Time

Training time optimization has mainly been achieved thanks to hardware accelerators (Intel, b; NVIDIA, c; Chen et al., 2014; Jouppi et al., 2017) hardware-specific libraries (Intel, a; NVIDIA, a), training strategies (Chilimbi et al., 2014; Krizhevsky, 2014; Li et al., 2014), neuron activation functions (Glorot et al., 2011; Nair & Hinton, 2010), and compilers (Google, c; Bergstra et al., 2010; Cyphers et al., 2018; Truong et al., 2016). Latte (Truong et al., 2016) showed preliminary results on how a compiler can achieve training time improvements through kernel fusion and loop tiling. Intel nGraph (Intel, c; Cyphers et al., 2018) leverages fusion, memory management and data reuse to accelerate training. Unlike our work, neither Latte nor nGraph explicitly target ensembles of DNNs: these optimization techniques are orthogonal to ours and could be combined with our approach in future works. Another interesting work is Wootz (Guan et al., 2019). Wootz is a system designed to reduce the time needed to prune convolutional neural networks. The speedups obtained by Wootz come from reducing the number of configurations to explore in the pruning design space as shown in Table 3 of (Guan et al., 2019). This, rather than reducing the training time of a single configuration, leads Wootz to find a good configuration faster. Even when ignoring their technical differences, CODE and Wootz are fundamentally orthogonal with respect to their goals: CODE aims at reducing the training time of an ensemble of networks while Wootz aims at reducing the number of configurations to explore for pruning (i.e., speeding up the pruning process). Interestingly, CODE and Wootz could be combined to exploit their orthogonality when an ensemble of convolutional neural networks needs to be pruned: Wootz could drive the pruning space exploration and CODE could reduce the time to train a single ensemble of such configuration.

5.4 Optimizing Inference Time

Inference time dictates the user experience and the use cases for which neural networks can be deployed. Because of this, inference optimization has been and still is the focus of a multitude of works. Backends (Apple; Intel, d; NVIDIA, b; Rotem et al., 2018), middle-ends (Chen et al., 2018; Cyphers et al., 2018), quantization (Gong et al., 2014; Han et al., 2015), and custom accelerators (Intel, b; NVIDIA, c; Jouppi et al., 2017; Reagen et al., 2016) are some of the proposed solutions to accelerate inference workloads. NNVM/TVM (Amazon; Apache; Chen et al., 2018) and nGraph (Cyphers et al., 2018) are two promising directions to address the inference problem. Our current approach does not target inference time optimization. Although we see great potential in combining current approaches with neuron level optimizations, we leave this as future work.

5.5 Software Stacks

Software stacks and frameworks are the de-facto standards to tackle deep learning workloads. TensorFlow (Abadi et al., 2016), Theano (Bastien et al., 2012; Bergstra et al., 2010), Caffe (Jia et al., 2014), MXNext (Chen et al., 2015), Microsoft Cognitive Toolkit (Microsoft; Yu et al., 2014), PaddlePaddle (Baidu), and PyTorch (Facebook, b) are some of these tools. The variety of frameworks and their programming models poses a threat to the usability and interoperability of such tools. ONNX (Facebook, a), nGraph (Cyphers et al., 2018), and NNVM/TVM (Amazon; Apache; Chen et al., 2018) try to address this problem by using connectors and bridges across different frameworks. We chose TensorFlow for our initial implementation due to its broad adoption. Nevertheless, our findings are orthogonal to the framework we used.

6 Conclusion

In this work we presented CODE, a compiler-based approach to train ensembles of DNNs. CODE managed to achieve the same output quality obtained by homogeneous ensembles of independently trained networks in a fraction of their training time and memory footprints. Our findings strongly suggest that the existence of redundancy within ensembles of DNNs deserves more attention. Redundancy not only negatively influences the final ensemble output quality but also hurts its training time. Our work targeted ensembles of neural networks and we believe there is more to be added to neuron level analyses. In its current iteration, CODE is capable of finding semantically-equivalence in neurons within fully connected layers across homogeneous networks. CODE can anyway be extended to support heterogeneous ensembles by means of functional-preserving architectural transformations. As an immediate future step, we aim to extend CODE to handle more sophisticated network architectures such as Inception (Szegedy et al., 2015) and ResNet (He et al., 2016) by adding support for neurons within convolutional layers to be part of common sub-networks.

Acknowledgments

We would like to thank the reviewers for the insightful feedback and comments they provided us.
Special thanks to Celestia Fang for proofreading multiple iterations of this manuscript.
This work was supported in part by NSF grant IIS-2006851.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pp. 265–283, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026899.

Aho, A. V., Sethi, R., and Ullman, J. D. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

Amazon. Introducing NNVM Compiler: A New Open End-to-End Compiler for AI Frameworks. URL https://amzn.to/2HIj3ws. Accessed: 2020-10-10.

Apache. TVM. URL https://tvm.apache.org/#about. Accessed: 2020-10-10.

Apple. CoreML. URL https://developer.apple.com/documentation/coreml. Accessed: 2020-10-10.

Baidu. PaddlePaddle. URL https://github.com/PaddlePaddle/Paddle. Accessed: 2020-10-10.

Bastien, F., Lamblin, P., Pascanu, R., Bergstra, J., Goodfellow, I., Bergeron, A., Bouchard, N., Warde-Farley, D., and Bengio, Y. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: A cpu and gpu math compiler in python. In *Proc. 9th Python in Science Conf*, volume 1, 2010.

Blalock, D., Ortiz, J. J. G., Frankle, J., and Guttag, J. What is the state of neural network pruning? *arXiv preprint arXiv:2003.03033*, 2020.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chen, T., Moreau, T., Jiang, Z., Shen, H., Yan, E., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: end-to-end compilation stack for deep learning. In *SysML Conference*, 2018.

Chen, Y., Luo, T., Liu, S., Zhang, S., He, L., Wang, J., Li, L., Chen, T., Xu, Z., Sun, N., and Temam, O. Dadiannao: A machine-learning supercomputer. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-47, pp. 609–622, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-6998-2. doi: 10.1109/MICRO.2014.58. URL http://dx.doi.org/10.1109/MICRO.2014.58.

Chilimbi, T. M., Suzue, Y., Apacible, J., and Kalyanaraman, K. Project adam: Building an efficient and scalable deep learning training system. In *OSDI*, volume 14, pp. 571–582, 2014.

Chiu, C., Sainath, T. N., Wu, Y., Prabhavalkar, R., Nguyen, P., Chen, Z., Kannan, A., Weiss, R. J., Rao, K., Gonina, K., Jaitly, N., Li, B., Chorowski, J., and Bacchiani, M. State-of-the-art speech recognition with sequence-to-sequence models. *CoRR*, abs/1712.01769, 2017. URL http://arxiv.org/abs/1712.01769.

Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., Kimball, R., Knight, J., Korovaiko, N., Vijay, V. K., Lao, Y., Lishka, C. R., Menon, J., Myers, J., Narayana, S. A., Procter, A., and Webb, T. J. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *CoRR*, abs/1801.08058, 2018. URL http://arxiv.org/abs/1801.08058.

Deng, L. and Platt, J. C. Ensemble deep learning for speech recognition. In *Fifteenth Annual Conference of the International Speech Communication Association*, 2014.

Facebook. ONNX, a. URL https://onnx.ai/. Accessed: 2020-10-10.

Facebook. PyTorch, b. URL https://pytorch.org/. Accessed: 2020-10-10.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*, 2014.

Google. TensorFlow AlexNet Model , a. URL https://github.com/tensorflow/models/blob/master/research/slim/nets/alexnet.py. Accessed: 2020-10-10.

Google. TensorFlow VGG Model , b. URL https://github.com/tensorflow/models/blob/master/research/slim/nets/vgg.py. Accessed: 2020-10-10.

Google. TensorFlow XLA, c. URL https://www.tensorflow.org/xla/. Accessed: 2020-10-10.

Guan, H., Shen, X., and Lim, S.-H. Wootz: A compiler-based framework for fast cnn pruning via composability. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pp. 717–730, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314652. URL http://doi.acm.org/10.1145/3314221.3314652.

Han, S., Mao, H., and Dally, W. J. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Hansen, L. K. and Salamon, P. Neural network ensembles. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (10):993–1001, 1990.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Hebb, D. O. *The organization of behavior*, volume 65. Wiley New York, 1949.

Hecht-Nielsen, R. Theory of the backpropagation neural network. In *Neural networks for perception*, pp. 65–93. Elsevier, 1992.

Hinton, G., Vinyals, O., and Dean, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

Hu, J., Shen, L., and Sun, G. Squeeze-and-excitation networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 7132–7141, 2018.

Huang, G., Li, Y., Pleiss, G., Liu, Z., Hopcroft, J. E., and Weinberger, K. Q. Snapshot ensembles: Train 1, get m for free. *arXiv preprint arXiv:1704.00109*, 2017.

ILSVRC. 2012, a. URL http://image-net.org/challenges/LSVRC/2012. Accessed: 2020-10-10.

ILSVRC. 2013, b. URL http://image-net.org/challenges/LSVRC/2013/results. Accessed: 2020-10-10.

ILSVRC. 2014, c. URL http://image-net.org/challenges/LSVRC/2014/results. Accessed: 2020-10-10.

ILSVRC. 2015, d. URL http://image-net.org/challenges/LSVRC/2015/results. Accessed: 2020-10-10.

ILSVRC. 2016, e. URL http://image-net.org/challenges/LSVRC/2016/results. Accessed: 2020-10-10.

ILSVRC. 2017, f. URL http://image-net.org/challenges/LSVRC/2017/results. Accessed: 2020-10-10.

Intel. MKL-DNN, a. URL https://github.com/intel/mkl-dnn. Accessed: 2020-10-10.

Intel. Neural Compute Stick, b. URL https://software.intel.com/en-us/neural-compute-stick. Accessed: 2020-10-10.

Intel. nGraph Library, c. URL https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html. Accessed: 2020-10-10.

Intel. plaidML, d. URL https://github.com/plaidml/plaidml. Accessed: 2020-10-10.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pp. 675–678. ACM, 2014.

Johnson, K. Facebook VP: AI has a compute dependency problem, 2019. URL https://bit.ly/3iQLtD8. Accessed: 2020-10-10.

Jouppi, N. P., Young, C., Patil, N., Patterson, D. A., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., Boyle, R., Cantin, P., Chao, C., Clark, C., Coriell, J., Daley, M., Dau, M., Dean, J., Gelb, B., Ghaemmaghami, T. V., Gottipati, R., Gulland, W., Hagmann, R., Ho, R. C., Hogberg, D., Hu, J., Hundt, R., Hurt, D., Ibarz, J., Jaffey, A., Jaworski, A., Kaplan, A., Khaitan, H., Koch, A., Kumar, N., Lacy, S., Laudon, J., Law, J., Le, D., Leary, C., Liu, Z., Lucke, K., Lundin, A., MacKean, G., Maggiore, A., Mahony, M., Miller, K., Nagarajan, R., Narayanaswami, R., Ni, R., Nix, K., Norrie, T., Omernick, M., Penukonda, N., Phelps, A., Ross, J., Salek, A., Samadiani, E., Severn, C., Sizikov, G., Snelham, M., Souter, J., Steinberg, D., Swing, A., Tan, M., Thorson, G., Tian, B., Toma, H., Tuttle, E., Vasudevan, V., Walter, R., Wang, W., Wilcox, E., and Yoon, D. H. In-datacenter performance analysis of a tensor processing unit. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pp. 1–12. IEEE, 2017.

Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., and Wu, Y. Exploring the limits of language modeling. *arXiv preprint arXiv:1602.02410*, 2016.

Kazemi, H. MNIST Network . URL https://github.com/Machinelearninguru/Deep_Learning/blob/master/TensorFlow/neural_networks/autoencoder/simple_autoencoder.py. Accessed: 2020-10-10.

Krizhevsky, A. AlexNet for CIFAR10, a. URL https://github.com/akrizhevsky/cuda-convnet2/blob/master/layers/layers-cifar10-11pct.cfg. Accessed: 2020-10-10.

Krizhevsky, A. AlexNet for ImageNet, b. URL https://github.com/akrizhevsky/cuda-convnet2/blob/master/layers/layer-params-imagenet-1gpu.cfg. Accessed: 2020-10-10.

Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *arXiv preprint arXiv:1404.5997*, 2014.

Krizhevsky, A., Nair, V., and Hinton, G. CIFAR Datasets . URL https://www.cs.toronto.edu/~kriz/cifar.html. Accessed: 2020-10-10.

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

LeCun, Y., Cortez, C., and Burges, C. C. MNIST Dataset . URL http://yann.lecun.com/exdb/mnist/. Accessed: 2020-10-10.

Lee, S., Purushwalkam, S., Cogswell, M., Crandall, D., and Batra, D. Why m heads are better than one: Training a diverse ensemble of deep networks. *arXiv preprint arXiv:1511.06314*, 2015.

Li, M., Andersen, D. G., Park, J. W., Smola, A. J., Ahmed, A., Josifovski, V., Long, J., Shekita, E. J., and Su, B.-Y. Scaling distributed machine learning with the parameter server. In *OSDI*, volume 14, pp. 583–598, 2014.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

Microsoft. Cognitive Toolkit. URL https://docs.microsoft.com/en-us/cognitive-toolkit/. Accessed: 2020-10-10.

Nair, V. and Hinton, G. E. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pp. 807–814, 2010.

NVIDIA. cuDNN, a. URL https://developer.nvidia.com/cudnn. Accessed: 2020-10-10.

NVIDIA. TensorRT, b. URL https://developer.nvidia.com/tensorrt. Accessed: 2020-10-10.

NVIDIA. Tesla V100 GPU, c. URL https://www.nvidia.com/en-us/data-center/tesla-v100/. Accessed: 2020-10-10.

Reagen, B., Whatmough, P., Adolf, R., Rama, S., Lee, H., Lee, S. K., Hernández-Lobato, J. M., Wei, G.-Y., and Brooks, D. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *ACM SIGARCH Computer Architecture News*, volume 44, pp. 267–278. IEEE Press, 2016.

Rice, J. A. *Mathematical statistics and data analysis*. Cengage Learning, 2006.

Rotem, N., Fix, J., Abdulrasool, S., Deng, S., Dzhabarov, R., Hegeman, J., Levenstein, R., Maher, B., Satish, N., Olesen, J., Park, J., Rakhov, A., and Smelyanskiy, M. Glow: Graph lowering compiler techniques for neural networks. *CoRR*, abs/1805.00907, 2018. URL http://arxiv.org/abs/1805.00907.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y.

Sergeev, A. and Del Balso, M. Horovod: fast and easy distributed deep learning in tensorflow. *arXiv preprint arXiv:1802.05799*, 2018.

Sharkey, A. J. *Combining artificial neural nets: ensemble and modular multi-net systems*. Springer Science & Business Media, 2012.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Sopyla, K. Plon.io MNIST Network . URL https://github.com/ksopyla/

tensorflow-mnist-convnets. Accessed: 2020-10-10.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1–9, 2015.

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *AAAI*, volume 4, pp. 12, 2017.

Truong, L., Barik, R., Totoni, E., Liu, H., Markley, C., Fox, A., and Shpeisman, T. Latte: a language, compiler, and runtime for elegant and efficient deep neural networks. *ACM SIGPLAN Notices*, 51(6):209–223, 2016.

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. Regularization of neural networks using dropconnect. In *International Conference on Machine Learning*, pp. 1058–1066, 2013.

Wang, F., Jiang, M., Qian, C., Yang, S., Li, C., Zhang, H., Wang, X., and Tang, X. Residual attention network for image classification. *arXiv preprint arXiv:1704.06904*, 2017.

Wasay, A., Hentschel, B., Liao, Y., Chen, S., and Idreos, S. Mothernets: Rapid deep ensemble learning. In *Proceedings of the Conference on Machine Learning and Systems (MLSys)*, 2020.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

Yu, D., Eversole, A., Seltzer, M., Yao, K., Kuchaiev, O., Zhang, Y., Seide, F., Huang, Z., Guenter, B., Wang, H., Droppo, J., Zweig, G., Rossbach, C., Gao, J., Stolcke, A., Currey, J., Slaney, M., Chen, G., Agarwal, A., Basoglu, C., Padmilac, M., Kamenev, A., Ivanov, V., Cypher, S., Parthasarathi, H., Mitra, B., Peng, B., and Huang, X. An introduction to computational networks and the computational network toolkit. Technical Report MSR-TR-2014-112, October 2014.

APPENDIX

A DEAD NEURONS AND CSN IMPACT

In our experiments, just removing dead neurons did not result in ensembles with higher output quality nor noticeable improvements in training time. The percentage of removed dead neurons is in fact extremely low with respect to the total number of neurons per network (in most cases we observed that it is less than 0.5% than the total number of neurons).
What follows are numbers our most relevant benchmarks for the results shown in this work.
In `VGG_I`, the number of dead neurons is 12 out of 13416 fully connected neurons ($\sim 0.09\%$). The number of neurons in the CSN is 1231 out of 13404 remaining neurons ($\sim 9.18\%$).
In `AlexNet_I`, the number of dead neurons is 17 out of 10472 fully connected neurons ($\sim 0.16\%$). The number of neurons in the CSN is 1584 out of 10455 remaining neurons ($\sim 15.15\%$).
There is a two order of magnitude difference between the number of dead neurons and the number of neurons included in the CSN. These numbers make us confident that CSNE is the step responsible for the results achieved by CODE.

B CSNE ROBUSTNESS

In our experiments we observed that CSNE is robust.
The two original networks, main and peer, have always been randomly initialized. Changing these two starting networks with two different randomly initialized ones always led CODE to obtain results in line with what has been shown in this work. To confirm that our approach does better than random neurons selection, during our preliminary studies, we tried to pick random neurons from the main network and added those to an initially empty CSN. We also tried to use CSNs made of randomly initialized neurons. In both of these cases the ensembles we obtained had substantially lower output quality than ensembles of independently trained networks. In short, CSNE does much better than random selection and is robust with respect to the two original networks used for the redundancy elimination phase described in Section 3.1.

C HIGHER QUALITY ENSEMBLES FOR UNBOUNDED TRAINING TIME

Our investigation suggests that common subnetworks (CSNs) contribute to this phenomenon in two different ways.

(i) CSNs can be considered as a way to initialize networks in a different and better way than random initialization. Deep neural networks are in fact sensitive to their initialization point. Using a CSN allows networks to start their training from a point in the whole solution space that is better than a randomly selected one.

(ii) The use of a CSN implies that the training of new networks will only be focused on the parameters that are outside of the CSN. This means that, during training only a subspace of the whole solution space will be explored.

Better initialization and focused training translate into ensembles with more useful diversity and heterogeneity than ensembles made of independently trained networks. These allow CODE to reach diminishing returns (in terms of ensemble output quality) much later than ensembles made of independently trained networks. We hope to quantify the relative impact of these two contributing factors in future works.

D DNN PRUNING

During our exploratory research we evaluated pruning with respect to CODE. While there is a wide variety of available pruning techniques (Blalock et al., 2020), pruning is generally used to reduce the numbers of parameters needed at inference time. Our initial results suggested that pruning would lead to sub-optimal output quality of the baselines, so we dropped those experiments. In our experience, pruning at training time also involves fine-tuning and/or additional training iterations which result in additional training time overhead. This overhead negatively impacts the usefulness of pruning as a baseline for fast training. These characteristics are in contrast with the main objective of CODE which is to reduce training time while retaining the same or better output quality of ensembles of independently trained DNNs.