A COLLABORATIVE APPROACH TO DEEP LEARNING

JUSTIN NOEL

Abstract. We introduce several techniques for training collaborative ensembles of deep neural networks. In addition to learning to solve an image classification problem, these collaborative ensembles also learn to reach a consensus on their predictions. As a consequence, each of the models in these ensembles learns to mimic a traditional ensemble, with corresponding improvements to performance.

Contents

Date: Sunday 4th June, 2017.

The author was partially supported by the DFG grants: NO 1175/1-1 and SFB 1085 - Higher Invariants, Regensburg.

1. Definition

1.1. **Project overview.** Recent advances in image classification have focused on constructing very deep and wide networks with 100+ layers [HZRS16a, SLJ+15]. While these networks represent the state of the art on standard benchmarks, training very deep and wide networks is extremely intensive computationally and introduces significant challenges to learning, typically due to vanishing gradients. Moreover, such models are often too slow at inference to be transferred to devices with limited computational resources, such as mobile devices.

Shallow and thin networks are much faster to train and to perform inference with, but typically learn models which generalize relatively poorly to testing data. Research has shown that for shallow and wide networks, this is not so much an artifact of the shallow model's inability to express the deeper model's predictions [HVD15, BC14, BCNM06], but rather, at least in part, the shallow model's tendency to overfit to the training data when given hard prediction targets.

On the other side, it has been shown that training medium depth networks can be made easier by providing easier to match targets [BLCW09, RBK+14].

2. Problem Statement

In this project we will investigate whether a *collaborative* approach to learning can improve on the above state of affairs. To do this, we will train various models to solve an image classification problem using a variety of cost functions which will lead the models to reach a consensus on their outputs. By pushing the models to reach a consensus, we will show that each of the individual models will be able to better approximate the performance of the ensemble.

A traditional ensemble typically outperforms each of its constituent models. Below we will see that collaboratively trained models learn to approximate the entire ensemble and outperform their corresponding models trained in isolation. The resulting collaborative ensembles have improved constituents and often improved ensemble performance. Since collaborative training is done concurrently with standard training, this improvement in performance comes at virtually no additional computational cost.

Below we will measure the generalization error in the usual fashion, via a test dataset and compare it to the outputs of models trained in the conventional, non-collaborative fashion. After examining the performance of a wide variety of collaboration methods. We will evaluate the training behavior by graphing the training and test errors of the best performing model.

More specifically, we will train two variations on the ResNet architecture [HZRS16a]. Here is the general layout of our implementation:

(1) Apply a batch normalization layer (BN) which learns to rescale the input.
(2) Apply a 3x3 convolutional layer with F filters, stride 1, and same padding.
(3) A composite of B Bricks which each consist of:
    (3.1) Applying R stacked instances of the following composite (where the input to a layer is x):
        (i) $o = \text{BN}(x)$
        (ii) $o = \text{Relu}(o) := \max(x, 0)$.
        (iii) Apply a 3x3 convolutional layer with same padding and the same number of filters and stride 1 to o. Set the output to o.
        (iv) Apply the previous three steps again to the o and set the output to o.
        (v) Return $x + o$.
    (3.2) For each of the first $B-1$ bricks, apply a dimension reduction and filter doubling layer which each consists of:
        (i) $o = \text{BN}(x)$
        (ii) $o = \text{Relu}(o) := \max(x, 0)$.

(iii) Apply a 2x2 convolutional layer with valid padding, double the number of filters of the input to the dimension reduction layer, and stride 2 to o. Set the output to o.
(iv) Apply the previous three steps again to the o and set the output to o.
(v) Return the sum of o and an application of 3(3.2)iii to x.

(3.3) Apply a global average pooling layer.
(3.4) Apply a fully connected layer with output in the unscaled logit space $\mathbb{R}^{10}$.
(3.5) Apply a softmax function to obtain the predictions.

As one can see, the general ResNet architecture can be easily scaled to varying depths by adjusting two parameters, the number of 'bricks' B and the number of repetitions R. With the exception of the last brick, each brick ends in a halving of each spatial dimension and a doubling of the number of filters. We adjust the width of the network by setting the number of filters F for the first convolutional layer. All models are 'fully convolutional' in the sense that they do not have any hidden fully connected layers. The resulting model will have $1+(B-1)(2R+2)+2R+1$ convolutional layers and end with a global average pooling operation and a linear map $\mathbb{R}^{2^{B}F} \to \mathbb{R}^{10}$ which is then composed with a softmax transformation.

The first model will be a 'shallow and thin' model (ST) with $B = 3, R = 1$ and $F = 8$, yielding a 10 layer model whose penultimate layer has dimension $2^6 = 64$. This model has approximately 50k learnable parameters. The second model will be a 'medium' (M) model with $B = 3, R = 2$, and $F = 16$, yielding a 16 layer model whose penultimate layer has dimension $2^7 = 128$. This model has approximately 280k learnable parameters.

Traditionally one might train these models consecutively and average their predictions in a suitable fashion to form the ensemble. We can train both models simultaneously by applying gradient descent to minimize the sum of the two cost functions for the two models:

$$\text{Cost}^{\text{ens}} = \text{Cost}^{ST} + \text{Cost}^{M}. \tag{2.1}$$

Typically, averaging the predictions reduces variance in the ensemble model's predictions and can improve generalization error.

Below we will compare this approach to a collaborative approach, where we will have the model 'collaborate' by replacing (2.1) with

$$\text{Cost}^{\text{coll}} = \text{Cost}^{\text{ens}} + \alpha \cdot \text{Collab}. \tag{2.2}$$

Here Collab will be the 'collaboration method' which will be one of three functions which measures the distance between the predictions produced by the models and α will be a 'collaboration weight' which will dictate how much we want to emphasize collaboration versus learning from the hard targets. Both the choice of the collaboration method and the collaboration weight are hyperparameters. After this rephrasing, we can regard this project as a simple hyperparameter search.

2.0.1. *Collaborative learning.* Before we make precise how our models will collaborate, we will recall how a neural network classifier is trained. Suppose that our classification problem is to find a function $f(x) = (f_1(x), \ldots, f_n(x))$ approximating the labeling $x \mapsto y(x) = (y_1(x), \ldots, y_n(x))$, where $y(x)$ is the one-hot encoding of a given classification of x into one of n-classes. In other words, if x belongs to the ith class, then $y_j(x) = \delta_{i,j}$.

Our models will have the form

$$f(x) = \text{Softmax}(g(x)) = \frac{1}{\sum e^{g_i(x)}}(g_1(x), \ldots, g_n(x)),$$

where $g(x)$ is the vector of unscaled logits our model produces and implicitly a function of a parameter vector θ. Such models are typically trained by attempting to minimize the cross-entropy of f with $y(x)$:

$$\begin{aligned} CE(y(x), f(x)) &= -\sum y_j(x) \ln f_j(x) \\ &= -\ln f_i(x) \\ &= \ln(\sum e^{g_j(x)}) - g_i(x). \end{aligned}$$

Under this rephrasing we are trying to learn a function g into $\mathbb{R}^n$ whose value will be as close to $+\infty$ as possible in the component corresponding to the positive classification and as close to $-\infty$ in each of the negative classification components.

Since neural networks are trained by variants of gradient descent it can be helpful to calculate the gradient flow of the cross entropy function with respect to a parameter θ_k:

$$\begin{aligned} \frac{\partial CE(y(x), f(x))}{\partial \theta_k} &= \sum_\ell \frac{\partial CE(y(x), f(x))}{\partial g_\ell(x)} \frac{\partial g_\ell(x)}{\partial \theta_k} \\ &= \sum_\ell \frac{e^{g_\ell(x)} \frac{\partial g_\ell(x)}{\partial \theta_k}}{\sum e^{g_m(x)}} - \delta_{\ell,i} \frac{\partial g_\ell(x)}{\partial \theta_k} \\ &= \sum_\ell \frac{\partial g_\ell(x)}{\partial \theta_k} (f_\ell(x) - \delta_{\ell,i}). \end{aligned}$$

In particular, we used that $\frac{\partial CE(y(x), f(x))}{\partial g_\ell(x)} = f_\ell(x) - \delta_{\ell,i}$. This means that the adjustment made to the ℓth unscaled logit to minimize the cross-entropy is proportional to the error in the ℓth prediction.

Note that the algorithm punishes all errors equally. For example, if the model produces the same value $g(x_1) = g(x_2)$ for two different training samples which should be labeled as class '7', then gradient descent will attempt to adjust the value of g in the same way in each component for each sample. However, if x_1 'really looks like' a '1' and x_2 is definitively a '7' then perhaps it would make sense to not try to adjust the first component of $g(x_1)$ as much as the second component; doing so may make it harder for the model to classify ones correctly (see Figure 3.4).

To understand this problem, so that we can solve it, consider the hypothesis that all possible inputs from our classification problem are pulled from some input space M (which would be a manifold according to the manifold hypothesis). Our model is attempting to learn how to label the connected components of our manifold according to some omniscient labeling scheme y. It will do this by finding a projection $M \to \mathbb{R}^n$ which in the ith coordinate has only the image of the ith labeled component of M landing in the positive numbers. Over time the model pushes the values to greater and greater extremes and thus separates M into components that are farther and farther away.

The problem with this approach is that it actually assumes the components of M are separable to begin with. In reality, it is not hard to image a sequence of seven's that gradually look more and more like a one (see Figure 3.4). If we take two arbitrarily close examples in M which are assigned different classifications by y the model learns to push these examples infinitely far from each other as fast as possible. A better approximation for our labeling problem should be fuzzier, where we assign to each point the degree the model should believe (in the Bayesian sense) a point lies in one class. This poses an easier task for optimization since not all classes need to be pushed so far apart and the model does not need to learn such a complicated embedding in order to separate them.

Now the problem is to find appropriate soft targets for optimization. As such targets are not given to us, we can regard this as an unsupervised learning problem. A solution to this unsupervised learning problem is approximated by the solution to the supervised classification problem: As the model learns to classify the images, it produces soft targets that reflect the similarities between different images that the labels do not.

So one can take two models, one which produces soft targets and another which trains to those soft targets. When this has been considered previously, this was done in two steps: first a very advanced model produces soft targets for a less complex model, which is then trained with these new target with the outcome that the less complex model outperforms the one trained on the hard targets (but still worse than the more advanced model). This produces a model which is much faster at inference than the more advanced model, but with slightly degraded performance.

We propose a similar implementation, but now we have the models learn simultaneously from each other. The models will partially target the hard targets, but the models will also learn to come to a consensus and produce approximately the same outputs. This will be implemented by changing the cost function as in (2.2) by setting $\text{Cost}^{\text{coll}}$ to be one of the following, where $f^{(1)}(x), f^{(2)}(x)$ are the prediction functions for the two models and $g^{(1)}(x), g^{(2)}(x)$ are their respective unscaled logits:

(1) (Symmetric cross entropy)

$$\text{Cost}^{CE} = -\left(\sum f_i^{(1)}(x) \ln f_i^{(2)}(x) + f_i^{(1)}(x) \ln f_i^{(1)}(x)\right).$$

(2) (L2-squared) $\text{Cost}^{L2} = \sum (g_i^{(1)}(x) - g_i^{(2)}(x))^2$

(3) (L1) $\text{Cost}^{L1} = \sum |g_i^{(1)}(x) - g_i^{(2)}(x)|$.

All of these cost functions are clearly non-negative. The first two obviously have global minima when the two models produce identical unscaled logits, so the cost functions are zero. Turning to the symmetric cross entropy function, when the two models produce identical predictions then the symmetric cross entropy is equal to double the entropy of the (common) model's predicted distribution. If these models were able to minimize the given cost function, then this distribution will have zero entropy and the model will have a global minimum. The upshot is that the collaborative models have theoretical global minima equal to those of their non-collaborative models and, a priori, collaboration does not necessarily obstruct learning the ideal function.

On the other hand, it is clear that when $\alpha \gg 0$ is very large, the backpropagation algorithm will prioritize minimizing the collaboration cost function over the ensemble cost function. This forces the individual models to try to solve the classification problem *while* producing identical predictions; a significantly more difficult task.

We will train a simpler model to have much better performance characteristics than it would otherwise and perhaps better than that of a model constructed by distillation/compression techniques [HVD15, BCNM06]. However, we will not be able to test this hypothesis here due to time and resource constraints (see Observation 4.1 below).

2.1. **Metrics.** Below we will evaluate the models by examining the average value of their respective error and cost functions on the training and test sets over the course of the training period. We will show that the test error rate for the ST and M models will nearly always be lower than those of the models trained in a traditional ensemble.

3. Analysis

3.1. **Data exploration.** We will base our analysis on 3 publicly available datasets: the MNIST, notMNIST, and CIFAR10 datasets.

3.1.1. *MNIST Dataset.* The first dataset is the MNIST dataset and is perhaps the urdataset for image classification benchmarks. It consists of 50k training and 10k test 28x28x1 images of handwritten digits (see Figure 3.1). As a standard dataset it will be easy to compare our performance to other published models. On this dataset, there are many approaches that can drop the error rate below 5 percent and most modern approaches should drop the error rate below 2 percent. As shown below our models will obtain an error rate of approximately 0.5%.

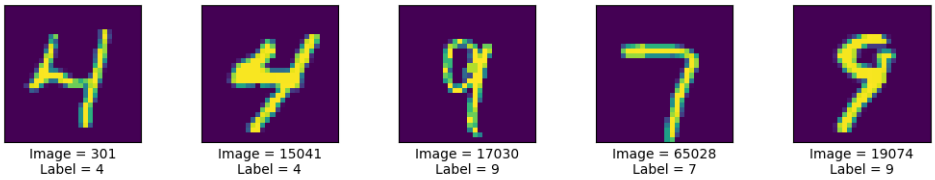


FIGURE 3.1. MNIST Sample Images

3.1.2. *notMNIST Dataset.* The notMNIST dataset consists of approximately 500k training and 19k test $(28, 28, 1)$ images of the characters 'a' through 'j' (see Figure 3.2). These characters are typeset with a wide variety of fonts. The training set has a roughly 6.5 percent misclassification rate, while the test set has a roughly 0.5 percent misclassification rate. Training against this dataset serves three functions: there are some additional challenges (memory constraints) involved in training a dataset of this size on a GPU, it appears to be a harder task than the MNIST classification problem, and it would be nice to see if our methods have a positive side effect on classifying problems with noisy and incorrect data. As such, we have made no attempt to clean up the data. Even without cleaning the data, we are still able to obtain error rates around 1.5%, which, since it appears that researchers have not seriously attempted to benchmark this dataset, might be state of the art.

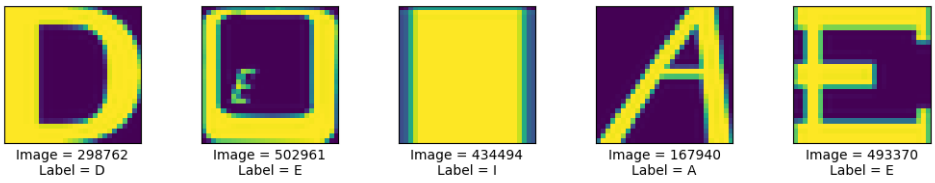
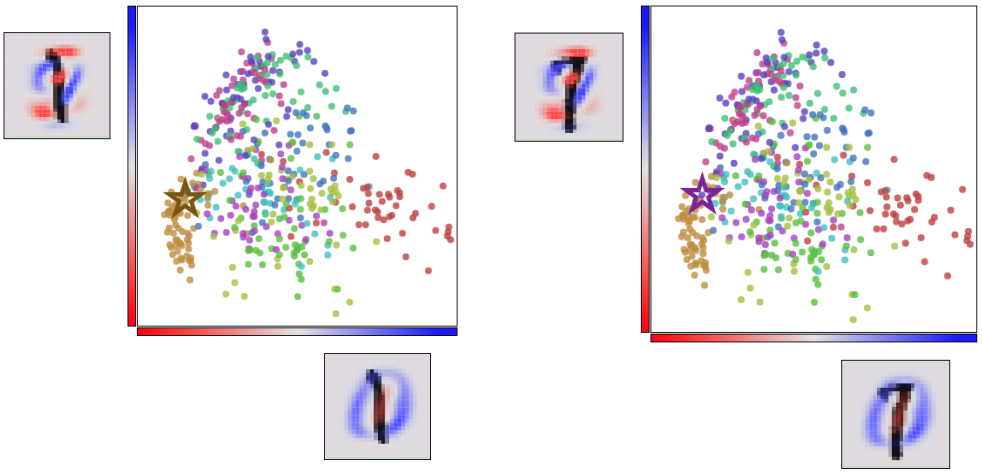


FIGURE 3.2. notMNIST Sample Images

3.1.3. *CIFAR10.* The CIFAR10 dataset consists of 50k training and 10k test $(32, 32, 3)$ images pulled from 10 categories (e.g., cat, dog, etc.) (see Figure 3.3). This dataset is the most challenging of the three and is still used to benchmark state of the art models. Our best results yield error rates of around 20%, which was state of the art in 2011, but far worse than what I expected to obtain with these models (see Remark 3.5).



(A) An MNIST '1'

(B) An MNIST '7'

FIGURE 3.4. Similar MNIST images with distinct classifications.

FIGURE 3.3. CIFAR10 Sample Images

One reason for choosing image classification benchmarks is that it has been shown that deep networks are especially suitable for such problems, yet it is desirable to obtain faster models at the inference stage.

3.1.4. *Further discussion.* All of these datasets are available for free online, and our provided code automatically downloads the datasets and puts them into the required format for our models.

3.2. **Exploratory visualization.** To illustrate our discussion from Section 2.0.1, we consider the MNIST dataset and perform a principal component projection to $\mathbb{R}^2$. This projection is chosen so that the projection to x-axis from the original dataset has the greatest variance in its image, while the projection to the y-axis produces the greatest variance in its image on the orthogonal complement of the projection to the x-axis. More informally this projection is chosen to spread out the data points as much as possible by maximizing the variance.

We have shamelessly stolen the visualization in Figure 3.4 from Chris Olah's blog [Ola]. Each of the subfigures contains an image of the 2D PCA projection of a subset of the MNIST dataset. In each of the two images we have drawn attention to a different data point (indicated by a star). These two points are very close to each other in the projection (and they are visually quite similar) although they correspond to samples with different classifications. One of the themes of this project is that a learned embedding which properly encodes the visual structure should

also map these two points to relatively nearby points in unscaled logit space, but standardized supervised learning techniques will attempt to push these points as far as part as possible, as if the two images were radically different.

As a quick aside, let us explain what the images on the axes represent. On each axis is a copy of the corresponding sample image. Overlaid on this image is a heat map indicating the weight a pixel receives under corresponding principal component, with the shades of blue indicating how positive the corresponding weight is and the shades of red indicating how negative a weight is. In other words, each projection is given by a linear map $\mathbb{R}^{28\cdot 28} \to \mathbb{R}$ which we can regard as a 28 by 28 matrix which is then viewed as an image which we superimpose on the sample image. This allows us to see how much each pixel contributes to the corresponding projection.

3.3. **Algorithms and Techniques.** As mentioned above, all of our networks are modeled on the ResNet architecture [HZRS16a]. We have made some minor variations to that model:

(1) We use the 'pre-activation' order of operations as studied here [HZRS16b]. This has been shown to improve the performance by a small amount.
(2) After each brick there is a halving in each spatial dimension and a doubling of the number of filters. In the published model this is performed by doing 1x1 convolutions with stride 2. Since this throws out 75% of the learned data without regard to the values, we have replaced this with a 2x2 convolutions with stride 2 and valid padding. As a consequence, these reductions in spatial dimensions can only be performed on tensors whose spatial dimensions are each divisible by 2.

Our choice of using the ResNet architecture was based on two reasons: 1) We wanted an architecture where it was straightforward to produce both deep and shallow variants and 2) we wanted an architecture which has been shown to have state of the art performance on competitive datasets.

Remark 3.5. Although the models I have trained all perform respectably on the given datasets, they are unable to match the state of the art performance on the CIFAR-10 dataset recorded in [HZRS16a] (the error rates for my models trained with data augmentation was closer to 15%, instead of the sub-10% error rates they obtain). Examining implementations online of this architecture, it appears that there are quite a few subtleties that are left out of the paper and that replicating their results requires making some choices and may have depended on the implementations of the algorithms they used (e.g., see the comments on their Git repository [HZRS]).

Another subtlety I noticed is that [HZRS16a] appears to not be counting the huge number of learned parameters in the batch normalization layers. I was under the impression that one counts parameters for a couple of reasons: First, parameters are one source of memory constraints and second, the number of parameters indicates the ability of the model to memorize a dataset. If this is the goal, the batch normalization parameters need to be counted for, at least, the first reason. Depending on how the batch normalization is implemented (e.g., if it can learn the identity function), then it should also be counted for the second reason.

Most striking, none of the models that I ran into could be trained with the published learning rate of 0.1, which I found to be far too high to be stable.

We will discuss the choices of hyperparameters in the implementation section.

3.4. **Benchmark.** As already indicated, we will test the performance of our collaboratively trained models against the benchmark set by the individual models when trained in an ensemble using the standard summary statistics (value of the cost/loss function and the error). The ensemble benchmark will give us reference values for how well the models perform when

trained individually. We will then compare this to how the models perform when we add in the collaboration cost functions.

4. Methodology

Before diving into the details of our methods we would like to point to a fact of life which became very clear to us while preparing this project.

Observation 4.1. Training deep neural networks is time consuming and expensive.

Observation 4.1 comes from the author's experience of spending $175 on Amazon web services to train these models. Roughly half of this was spent on tuning and debugging and the other half was spent training the final models which took 3 days on a P2xlarge instance (my first runs on G2 instances were prone to crashing due to the limited GPU memory, although this was most likely due to a bug that was later corrected). While many of our design decisions are probably suboptimal, they can typically be explained by Observation 4.1.

4.1. **Data Preprocessing.** Each of the models we have used begins with a batch normalization layer, which eliminates the need to normalize our data in advance. However there are still a few preprocessing steps we have implemented:

(1) We have shuffled all of our samples ahead of time. In the CIFAR10 and MNIST datasets we shuffled both the training and test sets together to ensure an even distribution across both. In the case of the notMNIST dataset, the test set has been cleaned ahead of time and has a drastically reduced misclassification rate. We would like to evaluate the effect of collaboration methods on noisier data so we have retained this property of the dataset.
(2) In order to preclude additional obfuscating factors, we did not perform data augmentation, although this seems to be critical for reaching state of the art performance.

5. Implementation

5.1. **Code structure.** We have implemented this project in Python using Tensorflow 1.1. We have written our own implementation of the above mentioned variation of the ResNet architecture. Because of the need to run so many different models and, at least during parameter tuning, to test against so many different choices of hyperparameters, we needed to write a Model class that made it simpler to consider the many varieties of ensembles. Each model consists of a Tensorflow graph for a given neural network/ensemble, a reference to a batch manager class, and commands to train the network, gather the statistics, and arrange for the statistics to be logged by Logger classes. The model's train functions perform regular backups of model checkpoints and pickles the accumulated statistical data.

The relatively short implementation of the ResNet network is contained in the 'resnet.py' file. The batch manager class gives the input pipeline and returns transformed batches.

The Logger class is responsible for logging and pickling data, as well as logging checkpoints. While the Model is responsible for generating statistics (cross entropy, error rate), the logger is responsible for storing this data in numpy arrays, generating pyplot graphs, and writing this data to disk at the end of each epoch. We wanted to compare both training and test statistics over the course of training, but we did not want to evaluate the entire test set on each batch, so we only provide statistics from the end of each epoch. The reported error rate is the error rate over that epoch and the reported cost is the average cost over that epoch. This produces significantly smoother graphs than one would normally see if we logged every batch.

Remark 5.1. Since the model is changing after each batch. It did not seem appropriate to average the statistics for the training data over each epoch. Instead we perform a running weighted average over each epoch. This provides better estimates in the early epochs when the

model's performance is rapidly improving. However on the MNIST dataset, the models eventually approximate state of the art performance and can run into long streaks of perfect predictions which cause the model to incorrectly return 0% error rates. In hindsight, I would have used statistics that were averaged over each epoch. We did not do this because of Observation 4.1.

The Model class has two subclasses: SingleModel and EnsembleModel. The SingleModel runs a single model and is a simplified form of the EnsembleModel which trains multiple models on the same task with or without collaboration.

The datasets file includes code for downloading, preprocessing, and generating sample images from the datasets. The code for handling the notMNIST dataset is directly taken from the Tensorflow example code and is not due to myself.

Finally, the analysis file gathers the pickled files, unpickles them, and stores the relevant data in a pandas data frame. There are also additional functions which generate the graphs and tables from the appendices.

5.2. **Hyperparameters.** One of the special challenges of this proposal was finding choices of hyperparameters that work reasonably well on both shallow and deep models. We have chosen to use L2 weight decay set to 0.0001 to further prevent potentially disastrous overfitting. We have also implemented a fairly simple learning rate scheduler: Once we are 50% through the training we cut the learning rate in half, we do this again at 75% and 90% of the training is completed. I was a bit surprised to see that cutting the learning rate had such a positive effect when used with the adaptive momentum optimizer (Adam) which dynamically adjusts the learning factors for each variable, but the effect is noticeable (one can see the drops in error rates at these spots). These cuts allow the optimizer to explore smaller crevices in the cost function graph and seem to improve the performance of the model.

Remark 5.2. The standard approach to learning rate scheduling is to pass a learning rate variable to the initialization of the optimizer and update this variable after each time step. However, in the case of the AdamOptimizer implementation in Tensorflow this causes the internal variables of the optimizer to be reinitialized after every time step and any learned momentum values to be lost. This drastically reduces the effectiveness of the optimizer.

We discovered that this effect can be avoided, by passing the learning rate to the optimizer as a placeholder value and not a variable.

All models were trained on mini-batches of size 64. We settled on this choice as the smallest mini-batch size that yielded relatively low variance cost and error statistics.

Remark 5.3. One potential source of the discrepancy between the performance of our models and the published reports of [HZRS16a] is that they use mini-batches of size 256 (divided over 8 GPUs)

For our collaborative ensembles $\{ST, M\}$, with logit (resp. softmax prediction) values on a sample x denoted by $l_i(x)$ (resp. $p_i(x)$), we considered the following collaborative cost functions:

- Symmetric cross entropy:

$$SCE(x) = -\sum_{i=0}^{10}(p_i^{ST}(x)\log p_i^M(x) + p_i^M(x)\log p_i^{ST}(x))$$

- Sum of squares distance:

$$L2(x) = \sum_{i=0}^{10}(l_i^M(x) - l_i^{ST}(x))^2$$

- Manhattan distance:

$$L1(x) = \sum_{i=0}^{10} |l_i^M(x) - l_i^{ST}(x)|$$

Sorry to spoil the surprise, but we chose to use the symmetric cross entropy function rather than the ordinary cross entropy because it is symmetric. This is more than mathematical aesthetics. Even though minimizing the cross-entropy function would push the models to come to a consensus, gradient descent will adjust the parameters of a model differently depending on whether or not we consider it to be the first or the second model. Symmetrizing the cross-entropy resolves this issue.

One reason to consider the sum of squares distance is that the gradient is proportional to the difference between the logits. So as the difference approaches zero the magnitude of the gradient decreases which will give the models a bit more room to try to match the hard prediction labels. While the Manhattan distance always has a gradient which is a unit multiple of the derivative of the logits (except when the distance is zero and the derivative is not uniquely defined).

Our implementation allows for ensembles of arbitrary size, in which case the collaborative cost functions are formed as a sum over consecutive models of functions of the above form. Due to Observation 4.1, we only apply this to ensembles of size 2 below.

We scale all of our collaboration cost functions by a hyperparameter $\alpha \in \{0.0001, 0.01, 0.1, 1.0\}$ called the collaboration weight and trained the following variations of our ensembles:

(1) Standard ensemble of ST and M models.
(2) Collaborative ensemble of ST and M models where $\alpha \in \{0.0001, 0.01.0.1, 1.0\}$:
    (2.1) Collaborative cost function $= \alpha \cdot SCE$.
    (2.2) Collaborative cost function $= \alpha \cdot L2$.
    (2.3) Collaborative cost function $= \alpha \cdot L1$.

Since each ensemble consists of two models and we additionally consider 3 different collaborative cost functions, each with one of four values of α, we have to train 13 different ensembles which in total involves training 26 models per dataset. We have to do this for each of the three datasets, which means we have to train 78 convolutional neural networks (see Observation 4.1).

For the MNIST and CIFAR10 datasets we trained each model for 30 epochs. On the notMNIST dataset we only trained for 15 epochs. Both of these decisions were based on Observation 4.1. I believe that there would be further gains if we trained longer since the Resnet models were trained for 3 times as long in [HZRS16a] on the CIFAR10 dataset.

5.3. **Refinement.** In the process of preparing the final models, we went through an extensive debugging and parameter tuning cycle. Debugging neural networks can be quite tricky, because broken code may not manifest itself except after extensive training, or it may just produce models that work, but not as well as they are supposed to. For example, a misplaced addition when composed with batch normalization can reduce the effectiveness of a model and cause the error rate to be at least 5 points higher [HZRS16b]. As another cautionary tale, we will mention that a misaligned tab caused our models to have about half as many layers; just enough to get acceptable performance, but quite short of what was expected.

Here are a few other lessons that I learned in the refinement process which are all closely related to Observation 4.1:

(1) Set random seeds. When making changes it is helpful if you can distinguish your changes from random chance.
(2) Unless absolutely necessary, debug with severely truncated datasets. If a bug is going to show up when your learning rate drops on the 30th epoch, it is best if you can see this in less than a day.

(3) Construct a simple model that is fast to train and obtains okay results and use this to debug the remainder of the code.
(4) Test your image classifiers on MNIST. If your error rate is above 3%, than something is probably wrong.
(5) Use name scopes and Tensorboard. Visualizing the computational graph is quite useful for debugging.

I will now mention a couple of the hyperparameters that were tuned in the early phase of this project. First we needed to establish the architectures for the shallow and thin (ST) model and the medium (M) model. We also needed to establish an acceptable mini-batch size. Both of these decisions were heavily influenced by Observation 4.1.

For the first task we wanted to find a shallow and thin model that was still able to obtain reasonable performance on the benchmarks (e.g., an error rate below 3% on MNIST) so that this model would have some hope of aping the more complex model. With the Resnet architecture it is easy to adjust the width and depth as mentioned above. Experiments with a starting filter depth of 4 ($F = 4$) showed that the model performed poorly regardless of the depth of the network, which is understandable for two simple reasons:

(1) The first convolutional layer only has 4 filter dimensions to map into. If the layer is supposed to reconstruct the image exactly and we have a color image (so 3 filters to begin with) this only leaves one independent dimension for a learned feature and minimal room to move the points in the sample space.
(2) Because the model's penultimate layer has dimension 32, this yields a fairly constrained space to construct the linearly embedding $\mathbb{R}^{32} \to \mathbb{R}^{10}$ to the unscaled logits which separates the classes.

Experiments with a feature depth of 8, yielded reasonable results, so we used this for the ST model and doubled it for the M model. The models from [HZRS16a] use 3 bricks ($B = 3$), so we decided to follow their lead and use this for both models. This only left the choice of how many convolutional layers to include in each brick which is encoded in the choice of R. We settled on $R = 1$ and $R = 2$ for the ST and M models respectively[1]. Again this choice was primarily dictated by Observation 4.1; if we were to expect near state of the art results, it would have made more sense to have set R to be at least 5 for the medium model.

In order to reduce the training time (see Observation 4.1), we experimented with very small mini-batches, working our way up from 16, by powers of 2. The small mini-batches worked, but produced extremely noisy statistics and we eventually settled on a mini-batch size of 64.

Do to Observation 4.1, we did not retrain the entire collection of 78 neural networks during this fine tuning process, so we do not have comparable data to relate initial and final solutions.

6. Results

6.1. **Model evaluation and validation.** After we trained our 78 models we have generated a lot of data which was pickled and squirreled away into 39 files. In the analysis file we gather the pickle data from each dataset and store it in a single pandas data frame to make it easy to slice up for our analysis. Since we have considered 3 different collaboration methods, each with 4 possible collaboration weights, on 3 different datasets, we will do a preliminary analysis to see which combinations of collaboration methods and weights produced the best outcomes in terms of test error. We will then proceed with a more detailed analysis of the better performing collaborative models.

[1]Although one can argue that a CNN with 10 convolutional layers is not 'shallow', this is probably a matter of perspective. It is definitely shallower than the 50+ layer models that are now in use.

Since we are considering so many models simultaneously we will focus our graphs and hence our attention on the final 10 training epochs and near the best performing models. The collaboratively trained models are all drawn in color and the traditionally trained models are in shades of gray.

We begin by considering the performance on the MNIST dataset in Figures A.1 and A.2 which are located in the appendix below. The careful reader will notice that there are not enough lines on the charts for the 'L2' and 'L1' collaborative models. This is because these methods with collaboration weight $\alpha = 1$ did so poorly that they do not show up on these graphs[2]. This illustrates that with large collaboration weights, the training process will primarily focus on making the models collaborate instead of fitting the data.

While this was expected, what was not expected is that this is not such a factor when we learn with the symmetric cross entropy method. A plausible explanation for this is by examining the gradients of these different cost functions. The gradients with respect to the symmetric cross entropy functions involve a sum of two terms, the first term we calculated above. This adjusts a logit for the first model at a rate proportional to the difference of the two models predictions. The second term is proportional to $p_i^{(1)}(1-p_i^{(1)})\log p_i^{(2)}$, which will become very small as the first model becomes confident in its predictions. So if the model has become reasonably confident in its predictions, the first term will dominate and we see that the gradients are essentially proportional to the difference between the two predictions (which will always be less than 1). This is in contrast to the gradients with respect to the L2 collaboration method which will be proportional to the difference between the *unscaled logits* which, by examining the values of these logits, will generally be larger than the difference between the predictions. The L1 norm will always be adjusting the unscaled logits by a factor of ± 1, which will again be larger in magnitude than the difference in the predictions. So the latter two collaboration methods typically emphasize collaboration more.

Looking over the charts in Figures A.1, A.2, A.3, A.4, A.5, and A.6 and the tables in Figures A.7, A.8, A.9, A.10, A.11, and A.12 (which are located in the appendix), there are a few important observations to make:

(1) Collaborative models usually outperform their traditionally trained counterparts: On *every* dataset at least 2 (and often more) of the collaborative models obtained lower test error than their counterparts trained in isolation.
(2) The collaboratively trained ST models can approximate the traditionally trained M models. This allows us to get comparable performance with 20% as many parameters and a threefold speedup.
(3) The traditionally trained ensembles out perform each of their constituents (as expected) and these models can be closely approximated by a single medium depth model that was trained in collaboration (obtaining a noticeable improvement in both memory and runtime over the ensemble model).
(4) All three collaboration methods yield improvements provided the collaboration rate is not too large and even for very small collaboration weights, we will often see an improvement.
(5) For symmetric cross entropy a collaboration weight of approximately 1 does well. For the L2 and L1 methods, smaller values generally do well.
(6) The performance of collaborative ensembles improves as the degree of the collaboration decreases. This makes sense since this will increase the variance in their predictions. But even with very low collaboration weights, the individual models still outperform their traditional counterparts, so the corresponding collaborative ensemble has better performance than the traditional ensemble.

[2]Adjusting the limits of the graphs to include these extra lines made the graphs illegible.

We will now further examine the collaborative model which generally yielded the best results: The L2 method with weight 0.01. We will examine the training behavior and see how the collaborative models perform with respect to overfitting. Examining Figures B.1, B.2, and B.3 in the appendix, we can make a few observations:

(1) Collaborative models have smoother learning curves than the traditionally trained models.
(2) On the MNIST dataset, the smoother learning curve appears to lead to better final results, By which we mean the model spends less time correcting for past mistakes.
(3) On the notMNIST dataset, whose training set has roughly a 6.5% misclassification rate[3], we see that the collaborative models are noticeably more resistant to overfitting, while still obtaining comparable results.
(4) On the CIFAR10 dataset, the collaborative models have very similar learning curves on the training set compared to their traditionally trained counterparts, yet they outperform these models on the test set.
(5) The models in the collaborative ensembles have more similar performance characteristics than those in traditional ensembles (as one should expect).

It appears that the collaborative learning has a noticeable regularization effect on the models which makes them more resistant to overfitting, this is similar to the behavior of ensembles (it is hard for distinct models to overfit the data in a way such that they generate highly correlated responses on new data). This may also be the source of the smoothing for the learning curves as the collaborative models are forced to make parameter adjustments that produce similar outputs which constrains the optimization problem in a constructive way (but this is just speculation).

6.2. **Justification.** We can see that for any of the above collaborative methods and most choices of weights our collaborative models modestly outperform traditionally trained ensembles. Since the only difference between the collaborative approach and the traditional approach is the choice of cost function, this performance enhancement comes at virtually no cost. The only restriction here, and it is an important one, is that it is necessary to optimize both all of the models in the collaborative ensemble simultaneously, which means all of the models need to fit into memory and this can be a significant constraint if we are trying to get multiple large networks to collaborate. On the other hand, it appears that collaboratively training a model with a much weaker model only *improves* both models. So adding smaller models into a collaboration should only require a comparatively small increase in memory, but noticeable improvements in performance.

7. Conclusion

7.1. **Free-Form Visualization.** Considering the above analysis, we conclude that are obvious and immediate benefits to training a collective of learning machines to solve a given task. If putting two neural networks together can produce such noticeable benefits, it is natural to ask what benefits we can get from ever larger and more varied collectives. After all, what is the worst thing that can happen? See Figure 7.1.

7.2. **Reflection.** We have trained a large number of convolutional neural networks of varying depth under a variety of collaborative training methods. For most reasonable choices of hyperparameters these collaborative ensembles modestly outperform traditional ensembles on three publicly available datasets. For larger collaboration weights, the collaborative ensembles are very close to those of the traditional ensembles. However, these collaboratively trained *individual* models outperform the traditionally trained counterparts by a more significant margin.

[3]As a consequence all models do better on the test set (which has a 0.5% misclassification rate) than on the training set.

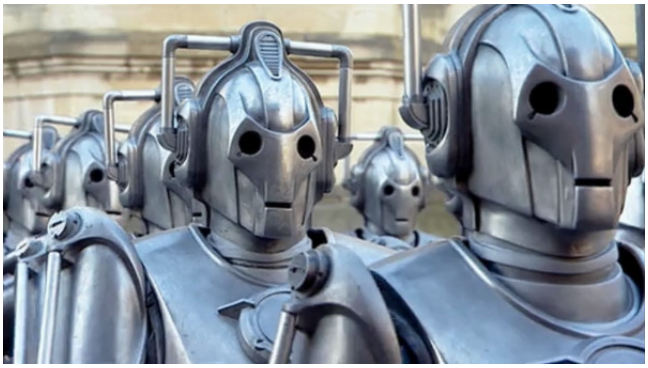


FIGURE 7.1. Do not fear the collective.

Moreover, for very small collaboration weights, the collaborative ensembles outperform the traditional ensembles. These performance enhancements come at virtually no cost provided that the computer has sufficient memory to train the models simultaneously.

Developing a project on this scale has been quite educational. Training so many models, gathering all of the resultant data, and combining it into understandable representations has required quite a bit more work than the 'quick-and-dirty' deep learning tutorials that one typically runs across[4]. I have personally experienced the difficulty of fine-tuning of hyperparameters on this scale (see Observation 4.1), as well as the manifold problems involved in debugging such networks.

Although, I believe that this project would benefit from further study, I view at as a success. Although it does not appear that the deeper models learn any faster when trained collaboratively, they actually do improve by working with a weaker model. To the author, this is a surprise and warrants further investigation.

7.3. **Improvement.** Now that we have identified reasonable collaboration methods and hyperparameters, it would be useful to further study these collaborative ensembles. For example:

(1) Do our results persist if we train the models for longer? If we add data augmentation?
(2) What happens if we change the collaboration weight over time?
(3) What if we consider asymmetric collaborations where some models are forced to collaborate more than others?
(4) What happens when we make the ensembles larger? The provided code is already designed for this generality.
(5) What happens when the ensembles consist of identical architectures? Do we still see such regularization effects?
(6) Can a Very Deep Model's performance be improved by working with such a shallow and thin model?
(7) We can easily separate the collaborative learning step from the training step. An important property of learning to collaborate is that *it does not require labeled training data.* In principle, a collaborative ensemble can learn to come to a consensus on any unlabeled dataset, even random noise. It would be interesting to see if mixing collaborative training and traditional training can produce even better outcomes and whether it is actually important to feed in similar data for the learning process.

[4]I would estimate that I have spent roughly 5 times as much time on this project as I have on all of the others combined.

Given the limited range of hyperparameters considered and the limited training time, I am confident that these models could be improved upon.

References

[BC14] Jimmy Ba and Rich Caruana. Do deep nets really need to be deep? In *Advances in neural information processing systems*, pages 2654–2662, 2014.

[BCNM06] Cristian Bucilu, Rich Caruana, and Alexandru Niculescu-Mizil. Model compression. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 535–541. ACM, 2006.

[BLCW09] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. Curriculum learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 41–48. ACM, 2009.

[HVD15] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[HZRS] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual networks.

[HZRS16a] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

[HZRS16b] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, pages 630–645. Springer, 2016.

[Ola] Christopher Olah. Visualizing mnist.

[RBK+14] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*, 2014.

[SLJ+15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

Appendix A. Comparison of methods and weights

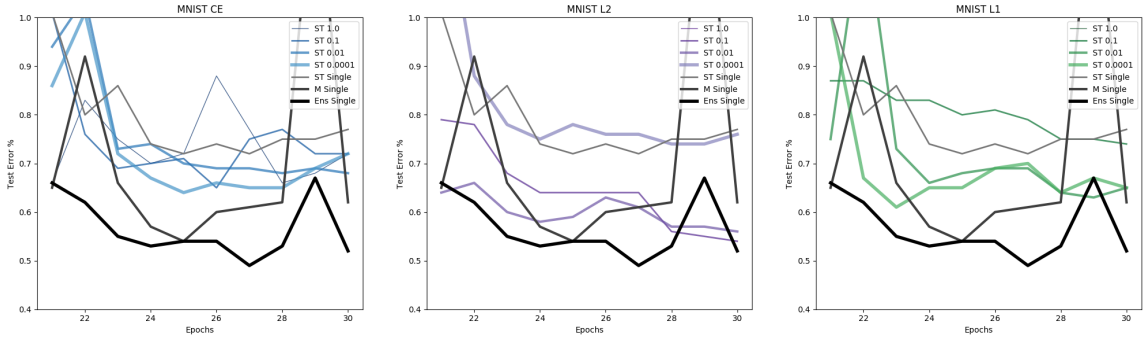


Figure A.1. MNIST Shallow and thin models

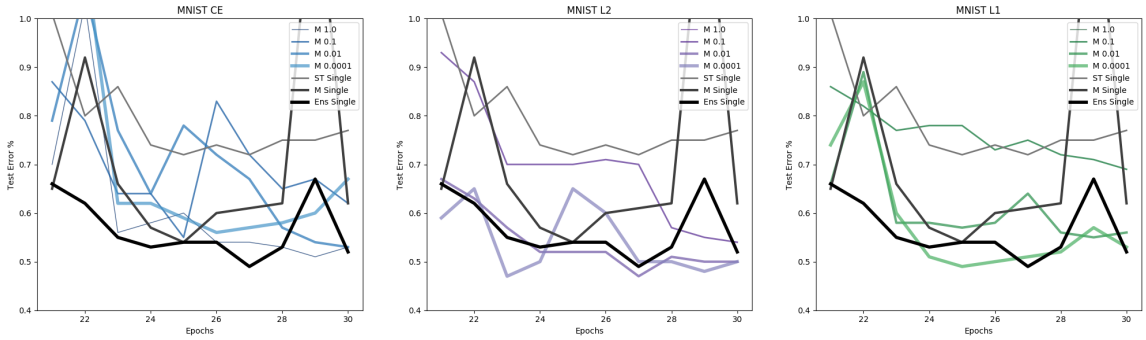


Figure A.2. MNIST Medium models

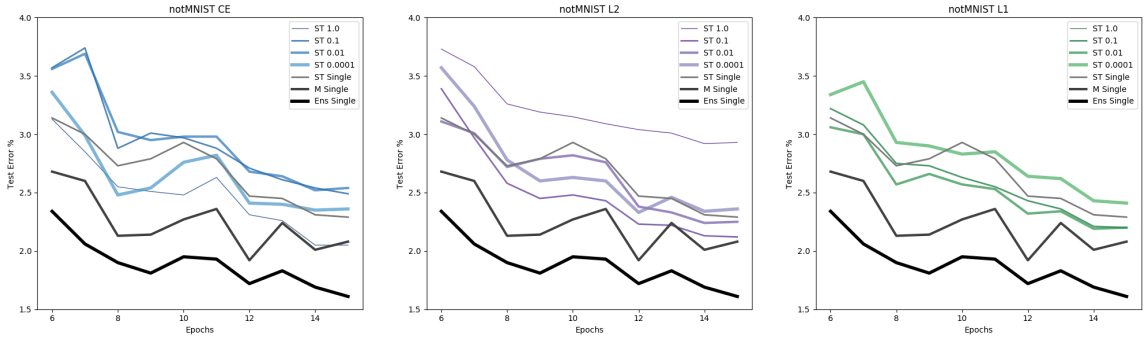


Figure A.3. notMNIST Shallow and thin models

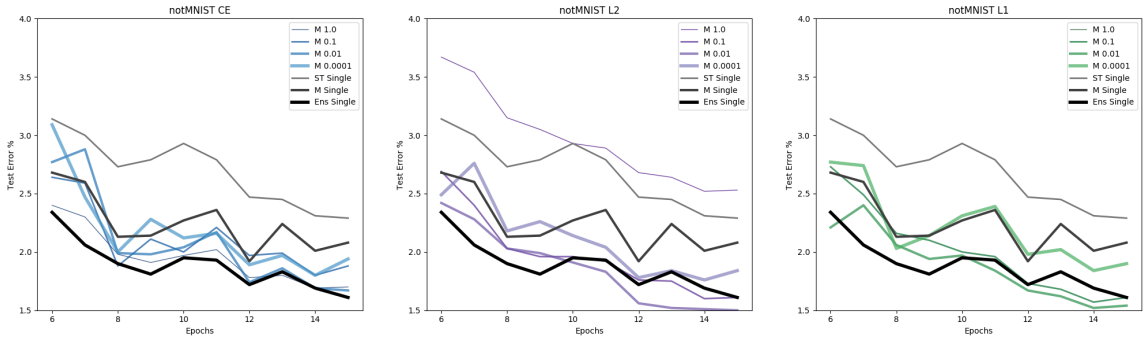


FIGURE A.4. notMNIST Medium models

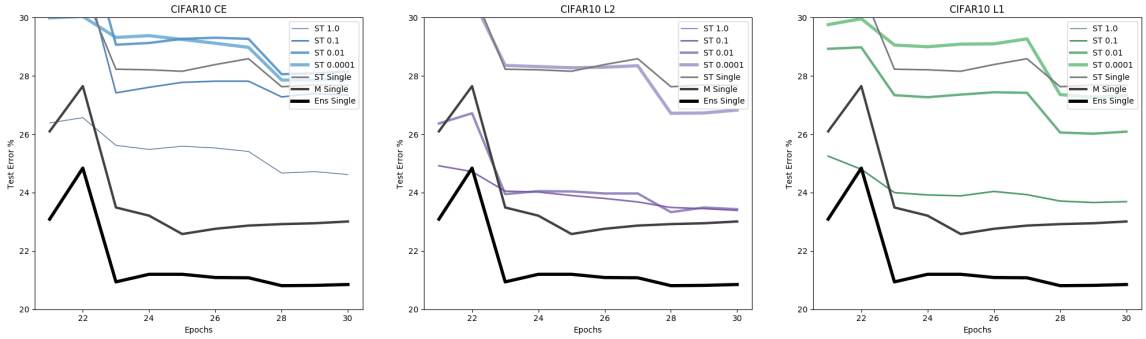


FIGURE A.5. CIFAR10 Shallow and thin models

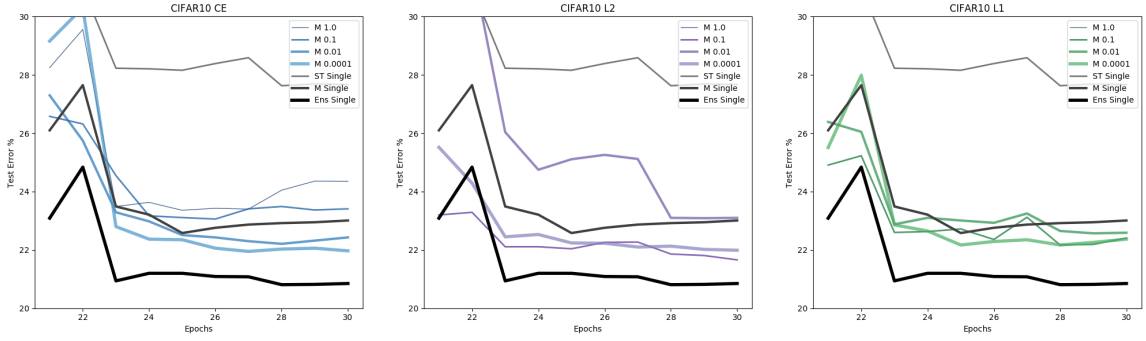


FIGURE A.6. CIFAR10 Medium models

A.1. **Graphs.**

A.2. **Tables.** Bold indicates best result.

ST Model	Test Error %	M Model	Test Error %	Ens Model	Test Error %
('CE', 1.0)	0.72	('CE', 1.0)	0.53	('CE', 1.0)	0.59
('CE', 0.1)	0.72	('CE', 0.1)	0.62	('CE', 0.1)	0.54
('CE', 0.01)	0.68	('CE', 0.01)	0.53	('CE', 0.01)	0.52
('CE', 0.0001)	0.72	('CE', 0.0001)	0.67	('CE', 0.0001)	0.61
('L1', 1.0)	2.75	('L1', 1.0)	2.75	('L1', 1.0)	2.74
('L1', 0.1)	0.74	('L1', 0.1)	0.69	('L1', 0.1)	0.73
('L1', 0.01)	0.65	('L1', 0.01)	0.56	('L1', 0.01)	0.59
('L1', 0.0001)	0.65	('L1', 0.0001)	0.53	('L1', 0.0001)	0.53
('L2', 1.0)	1.38	('L2', 1.0)	1.33	('L2', 1.0)	1.37
('L2', 0.1)	**0.54**	('L2', 0.1)	0.54	('L2', 0.1)	0.56
('L2', 0.01)	0.56	('L2', 0.01)	**0.5**	('L2', 0.01)	**0.5**
('L2', 0.0001)	0.76	('L2', 0.0001)	**0.5**	('L2', 0.0001)	0.57
('Single', 1.0)	0.77	('Single', 1.0)	0.62	('Single', 1.0)	0.52

FIGURE A.7. MNIST Final Test Error Results

ST Model	Test Cost	M Model	Test Cost	Ens Model	Test Cost
('CE', 1.0)	0.034945	('CE', 1.0)	0.026722	('CE', 1.0)	0.0191731
('CE', 0.1)	0.0421644	('CE', 0.1)	0.0378052	('CE', 0.1)	0.0248691
('CE', 0.01)	0.04444	('CE', 0.01)	0.0440946	('CE', 0.01)	0.0294754
('CE', 0.0001)	0.0468188	('CE', 0.0001)	0.0480394	('CE', 0.0001)	0.0329928
('L1', 1.0)	0.156449	('L1', 1.0)	0.179248	('L1', 1.0)	0.137231
('L1', 0.1)	0.0410959	('L1', 0.1)	0.0429026	('L1', 0.1)	0.0271439
('L1', 0.01)	0.0280316	('L1', 0.01)	0.0220588	('L1', 0.01)	0.0183511
('L1', 0.0001)	0.0324634	('L1', 0.0001)	0.0300227	('L1', 0.0001)	0.0218571
('L2', 1.0)	0.0729461	('L2', 1.0)	0.0850099	('L2', 1.0)	0.0549106
('L2', 0.1)	0.0316807	('L2', 0.1)	0.0308176	('L2', 0.1)	0.0200527
('L2', 0.01)	**0.0257211**	('L2', 0.01)	**0.0197911**	('L2', 0.01)	**0.0168002**
('L2', 0.0001)	0.0316473	('L2', 0.0001)	0.0261167	('L2', 0.0001)	0.0201273
('Single', 1.0)	0.0459429	('Single', 1.0)	0.0388529	('Single', 1.0)	0.0286906

FIGURE A.8. MNIST Final Test Cost Results

ST Model	Test Error %	M Model	Test Error %	Ens Model	Test Error %
('CE', 1.0)	**2.05**	('CE', 1.0)	1.7	('CE', 1.0)	1.67
('CE', 0.1)	2.49	('CE', 0.1)	1.88	('CE', 0.1)	1.74
('CE', 0.01)	2.54	('CE', 0.01)	1.67	('CE', 0.01)	1.61
('CE', 0.0001)	2.36	('CE', 0.0001)	1.94	('CE', 0.0001)	1.61
('L1', 1.0)	4.43	('L1', 1.0)	4.31	('L1', 1.0)	4.37
('L1', 0.1)	2.2	('L1', 0.1)	1.61	('L1', 0.1)	1.78
('L1', 0.01)	2.2	('L1', 0.01)	1.54	('L1', 0.01)	1.58
('L1', 0.0001)	2.41	('L1', 0.0001)	1.9	('L1', 0.0001)	1.59
('L2', 1.0)	2.93	('L2', 1.0)	2.53	('L2', 1.0)	2.7
('L2', 0.1)	2.12	('L2', 0.1)	1.61	('L2', 0.1)	1.74
('L2', 0.01)	2.25	('L2', 0.01)	**1.5**	('L2', 0.01)	**1.55**
('L2', 0.0001)	2.36	('L2', 0.0001)	1.84	('L2', 0.0001)	1.57
('Single', 1.0)	2.29	('Single', 1.0)	2.08	('Single', 1.0)	1.61

FIGURE A.9. notMNIST Final Test Error Results

ST Model	Test Cost	M Model	Test Cost	Ens Model	Test Cost
('CE', 1.0)	0.0912072	('CE', 1.0)	0.0729853	('CE', 1.0)	0.0597912
('CE', 0.1)	0.10222	('CE', 0.1)	0.0886137	('CE', 0.1)	0.0566993
('CE', 0.01)	0.107765	('CE', 0.01)	0.10884	('CE', 0.01)	0.0622045
('CE', 0.0001)	0.0975916	('CE', 0.0001)	0.113915	('CE', 0.0001)	0.0580449
('L1', 1.0)	0.189134	('L1', 1.0)	0.191429	('L1', 1.0)	0.171096
('L1', 0.1)	0.0967725	('L1', 0.1)	0.0726474	('L1', 0.1)	0.0675368
('L1', 0.01)	0.0909284	('L1', 0.01)	0.0764119	('L1', 0.01)	**0.053546**
('L1', 0.0001)	0.103207	('L1', 0.0001)	0.106903	('L1', 0.0001)	0.0596633
('L2', 1.0)	0.119305	('L2', 1.0)	0.113529	('L2', 1.0)	0.10414
('L2', 0.1)	0.0901654	('L2', 0.1)	0.0722657	('L2', 0.1)	0.0662774
('L2', 0.01)	**0.0891081**	('L2', 0.01)	**0.0712102**	('L2', 0.01)	0.055066
('L2', 0.0001)	0.0979867	('L2', 0.0001)	0.100407	('L2', 0.0001)	0.056186
('Single', 1.0)	0.0973864	('Single', 1.0)	0.122331	('Single', 1.0)	0.0619581

FIGURE A.10. notMNIST Final Test Cost Results

ST Model	Test Error %	M Model	Test Error %	Ens Model	Test Error %
('CE', 1.0)	24.62	('CE', 1.0)	24.35	('CE', 1.0)	21.6
('CE', 0.1)	27.34	('CE', 0.1)	23.41	('CE', 0.1)	20.85
('CE', 0.01)	28.15	('CE', 0.01)	22.43	('CE', 0.01)	20.42
('CE', 0.0001)	27.9	('CE', 0.0001)	21.97	('CE', 0.0001)	20.01
('L1', 1.0)	45.74	('L1', 1.0)	45.12	('L1', 1.0)	45.42
('L1', 0.1)	23.69	('L1', 0.1)	22.41	('L1', 0.1)	20.93
('L1', 0.01)	26.09	('L1', 0.01)	22.59	('L1', 0.01)	20.86
('L1', 0.0001)	27.33	('L1', 0.0001)	22.37	('L1', 0.0001)	20.16
('L2', 1.0)	34.87	('L2', 1.0)	33.53	('L2', 1.0)	34.15
('L2', 0.1)	**23.39**	('L2', 0.1)	**21.66**	('L2', 0.1)	21.12
('L2', 0.01)	23.43	('L2', 0.01)	23.1	('L2', 0.01)	20.32
('L2', 0.0001)	26.83	('L2', 0.0001)	21.99	('L2', 0.0001)	**19.72**
('Single', 1.0)	27.67	('Single', 1.0)	23.01	('Single', 1.0)	20.85

FIGURE A.11. CIFAR10 Final Test Error Results

ST Model	Test Cost	M Model	Test Cost	Ens Model	Test Cost
('CE', 1.0)	0.774816	('CE', 1.0)	0.787899	('CE', 1.0)	0.627583
('CE', 0.1)	0.949555	('CE', 0.1)	0.870409	('CE', 0.1)	0.673286
('CE', 0.01)	0.963046	('CE', 0.01)	1.11529	('CE', 0.01)	0.742132
('CE', 0.0001)	0.957336	('CE', 0.0001)	1.77734	('CE', 0.0001)	0.991118
('L1', 1.0)	1.31941	('L1', 1.0)	1.33951	('L1', 1.0)	1.29754
('L1', 0.1)	0.704402	('L1', 0.1)	0.705127	('L1', 0.1)	0.612594
('L1', 0.01)	0.843182	('L1', 0.01)	0.826856	('L1', 0.01)	0.638807
('L1', 0.0001)	0.960852	('L1', 0.0001)	1.43472	('L1', 0.0001)	0.846999
('L2', 1.0)	1.02301	('L2', 1.0)	1.01763	('L2', 1.0)	0.987183
('L2', 0.1)	**0.696001**	('L2', 0.1)	**0.689528**	('L2', 0.1)	0.624529
('L2', 0.01)	0.706045	('L2', 0.01)	0.766554	('L2', 0.01)	**0.595218**
('L2', 0.0001)	0.925666	('L2', 0.0001)	0.990678	('L2', 0.0001)	0.677036
('Single', 1.0)	0.96121	('Single', 1.0)	1.94576	('Single', 1.0)	1.07356

FIGURE A.12. CIFAR10 Final Test Cost Results

Appendix B. Detailed comparison

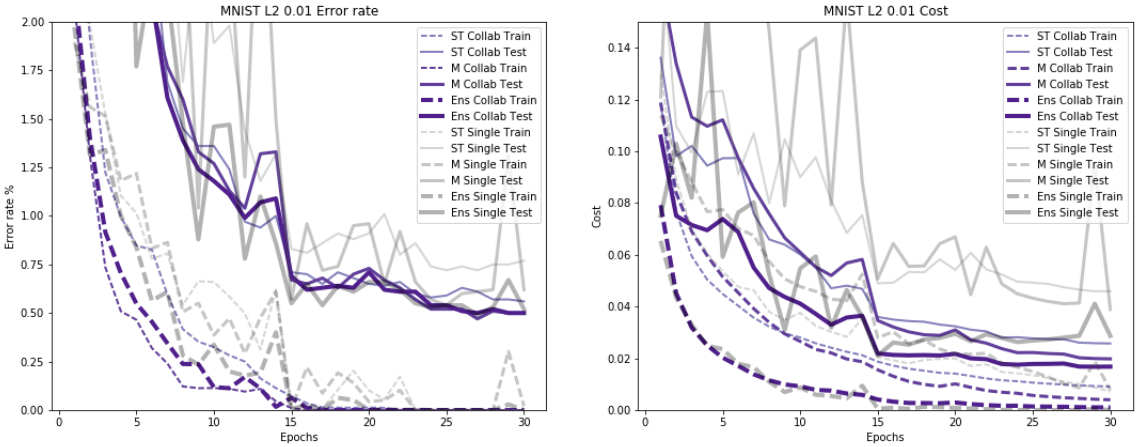


Figure B.1. MNIST Collab model

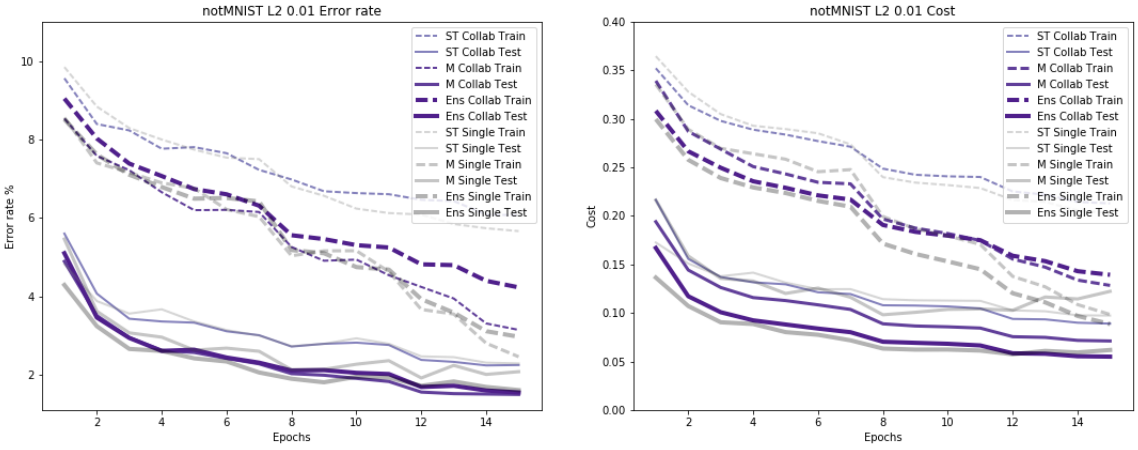


Figure B.2. notMNIST Collab model

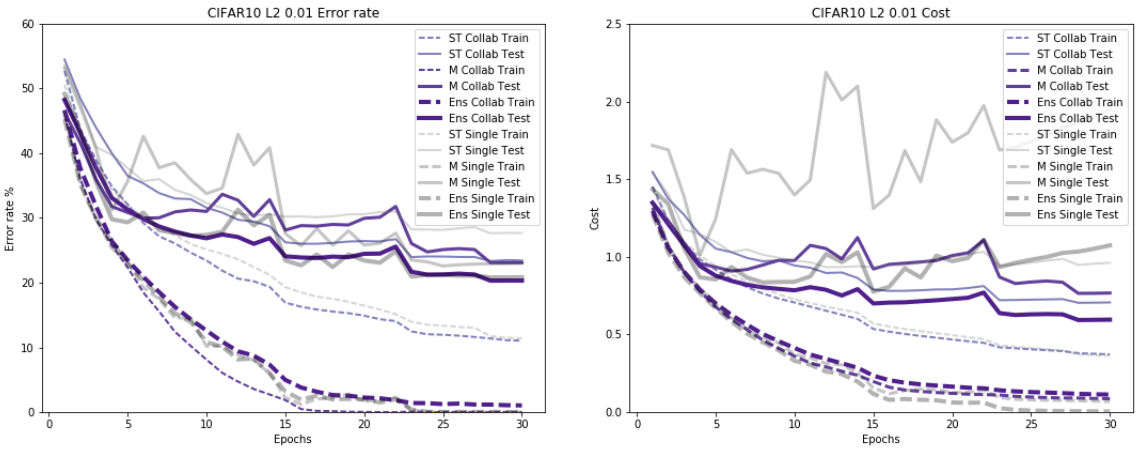


FIGURE B.3. CIFAR10 Collab model

UNIVERSITY OF REGENSBURG, NWF I - MATHEMATIK; REGENSBURG, GERMANY
E-mail address: justin.noel@mathematik.uni-regensburg.de
URL: http://nullplug.org