Thesis Proposal:
Efficient Deep Learning

Anders Øland
(anderso@cs.cmu.edu)

April 8, 2018

Abstract

It is well-known that training deep neural networks is a computationally intensive process. Given the proven utility of deep learning, efficiency is thus an important concern. In the thesis, we will review our previous related work on reducing the communication overhead in distributed deep learning, speeding up learning by boosting the error gradients, and how to implement neural networks efficiently on GPUs. We propose a new and simple method for layer-wise training of deep neural networks, that allows for the incremental addition of layers, such that the final architecture need not be known in advance. In conjunction, we explore a novel optimization method for non-linear regression problems, that uses error deltas instead of gradients, and which performs very well in simulations. We will investigate how this algorithm compares to gradient descent, and how it may be applied to training neural networks. Our end-goal is to make deep network training faster, simpler, and less reliant on expert knowledge.

1 Introduction

The recent second (or third) coming of neural networks, now known as deep learning (DL), has undeniably yielded some impressive results in computer vision, speech recognition, and multiple other fields. This has prompted some to name AI the fourth industrial revolution. Thus, massive amounts of effort, money, and energy are being put into the field, with deep learning currently receiving most of the attention. This means, that a focus on efficiency is now more important than ever. In the thesis, we will review our previous work on speeding up distributed deep learning, improving convergence rates by error gradient boosting, and strategies for efficient implementations of neural networks using GPUs (graphics processing units). We shall continue this line of work by proposing a new and simple method for layer-wise training of deep neural networks. This constructive training scheme allows for the incremental addition of layers, such that the final architecture need not be known in advance. The method is also less reliant on hyper-parameter tuning, and advanced methods such as batch normalization. In conjunction, we explore a novel optimization method for non-linear regression problems. This algorithm does not rely on error gradients but rather on error deltas and performs very well
in simulations. We will investigate how this algorithm compares to gradient descent, and
how it may be applied to training neural networks. Thus, with this work we aim to make
deep network training faster, simpler, and less reliant on expert knowledge.

1.1 Our Focus

The past decade has seen an explosion in methods, architectures, and algorithms related
to deep neural networks (DNNs). Given the wide range of learning models and tech-
niques, in this work we will limit our focus to the most common and influential ones,
namely: strictly feed-forward (non-recurrent) models with fully-connected and/or convo-
olutional layers trained with backpropagation and stochastic gradient descent (SGD) with
momentum. When relevant, we may choose to briefly expand our focus to include other
methods or architectures.

In our discussion and comparison of methods, we will consider performance aspects
such as:

1. Error rates and accuracy.
2. Convergence rates.
3. Hyper-parameters & expert knowledge required.
5. Memory usage.
6. Ease of implementation.
7. Parallelization properties.
8. Energy consumption.

Note, that this is not to be taken as a strict systematic comparison on the above criteria,
but rather a guideline for our discussion. Where applicable, we will use statistics, such
as medians and variances, rather than one-off results to evaluate our results.

2 Neural Networks

A basic feed-forward neural network (FNN) is a nested function consisting of a sequence of
pairs of affine transformations and non-linear functions; each pair is referred to as a layer.
For example, a two-layer FNN would be $y = f_2 \circ f_1 = f_2(f_1(x; W_1, b_1); W_2, b_2)$, where each
layer $f_i$ is $f_i(x; W_i, b_i, \sigma_i) = \sigma_i(xW_i + b_i)$. The matrices $W_i$ and the vectors $b_i$ are called the
weights and the bias of a layer, respectively. The functions $\sigma_i(\cdot)$ are generally non-linear,
and referred to as the activation function (AKA the non-linearity or squasher) of a layer.
A FNN with $k$ layers is thus a $(k-1)$-nested function $y = f_k \circ f_{k-1} \circ \cdots \circ f_1$ that, when
trained, learns some mapping from the set of inputs $X$ to the target outputs $Y$. 
Figure 1 shows a 3-4-2 FNN. That is, a two-layer neural network (we don’t include the input layer in the count) with three inputs, one hidden layer with four neurons, and two outputs; we say that the width of the input layer is 3, the width of the hidden layer is 4, and the width of the output layer is 2. To this FNN, a single input example \( x \) would be a \( 1 \times 3 \) vector, \( W_1 \) a \( 3 \times 4 \) matrix, \( b_1 \) a \( 1 \times 4 \) vector, \( W_2 \) a \( 4 \times 2 \) matrix, \( b_2 \) a \( 1 \times 2 \) vector, and the output \( y \) a \( 1 \times 2 \) vector.

![Figure 1: A 3-4-2 Feed-Forward Neural Network](image)

Figure 1: A 3-4-2 Feed-Forward Neural Network

It is standard notation in the literature to omit the bias term which is then assumed to be captured by adding an extra component of 1 to each input vector, such that \( x = \{1, x_1, \ldots, x_n\} \). We will adopt this notation from here on. Thus, each layer is now defined by \( f_i(x; W_i, \sigma_i) = \sigma_i(xW_i). \)

**Definition 1** (Neural Network). A \( k \)-layer feed-forward neural network (NN) with layer widths, \( d_i \in \mathbb{N}, i = 1 \ldots k \), is a nested function \( f : \mathbb{R}^n \mapsto \mathbb{R}^p \) defined by the recurrence relation

\[
f_i = \sigma_i(\tau_i(f_{i-1})), f_0 = x
\]

where \( \sigma_i : \mathbb{R} \mapsto \mathbb{R} \) is generally non-linear, \( \tau_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i} \) is a linear transformation, and \( d_0 = n, d_k = p \).

For so-called fully-connected layers \( \tau_i \) is an affine transformation. In a convolutional neural network (CNN) the \( \tau_i \)'s can also be the convolution or pooling operations.

In our discussion, we shall be referring to the data representations in a NN:

**Definition 2** (NN Representations). In a \( k \)-layer neural network, \( f : X \mapsto \hat{Y} \), the representation \( A_i = \tau_i(f_{i-1}) \) is the preimage of \( \sigma_i \), and the representation \( Z_i = \sigma_i(A_i) \) is the image of \( \sigma_i, i = 0 \ldots k \), where \( Z_0 = A_0 = X \) and \( Z_k = \hat{Y} \).

Thus, for a FNN we have the hidden representations (i.e. \( 0 < i < k \)), \( A_i = Z_{i-1}W_i \) and \( Z_i = \sigma_i(A_i) \). We will also be referring to the architecture of a neural network. That is, the set of all properties required to sufficiently describe a particular network, such as the number of neurons and the activation functions for each layer. This is the mathematical equivalent to the Protocol Buffer- and YAML-based formats used by many deep learning software frameworks. For example, the architecture, \( \Psi \), for our 3-4-2 FNN above would be
\[ \Psi = (\{4, \sigma_1\}, \{2, \sigma_2\}) \]. Here, and for all FNNs, the linear transformation associated with each layer is implied, namely the affine transformation. For more complex architectures, involving e.g. convolutions, pooling, or recurrences, it would need to be specified.

**Definition 3 (Architecture).** The architecture of a k-layer neural network is the tuple \( \Psi = (\psi_1, \psi_2, \ldots, \psi_k) \), where each \( \psi_i \in \Psi \) is the set of all properties associated with the \( i \)'th layer.

We will not concern ourselves with listing all the possible layer types and their associated properties. We will simply assume that all relevant properties are included in any given configuration. Obviously, we will also need to talk about learning problems. Hence, for the sake of consistency and nomenclature, let us make clear what exactly we mean by that.

**Definition 4 (Learning Problem).** Given a dataset \( D(X, Y) \), \( X \in \mathbb{R}^{m \times n} \), \( Y \in \mathbb{R}^{m \times p} \), and a loss function, \( L(Y, \hat{Y}) \), the task of approximating a function \( f : X \mapsto \hat{Y} \) such that \( L \) is minimized is referred to as a learning problem.

We could say that the dataset \( D \) captures the learning problem. In the following, we will use the terms dataset and learning problem interchangeably.

### 3 State of the Art

The one field where DL has been most influential is probably computer vision. Thus, we will mainly focus on the progress made on the most standard benchmarks in this field. Possibly the most important paper, the one that ignited the DL revolution, is \cite{14}. It has been cited over 17,000 times. This paper presented a huge leap in the classification accuracy on the ImageNet dataset of 1.3 million images in 1000 categories. As in \cite{14}, the state of the art is still CNNs with rectified linear units (ReLUs) in the hidden layers, trained with SGD + momentum. The main results in the literature since then are methods that attempt to overcome the short-comings of the backpropagation algorithm. The most infamous short-coming being the problem of vanishing gradients \cite{9}. In our opinion, the four most influential methods are:

1. Rectified linear units (ReLUs) that allow faster learning by *not* saturating and thus preventing error gradients from getting unnecessarily small during training.

2. Dropout \cite{8} which regularizes the training process by randomly dropping out activations (setting them to zero).

3. Batch normalization (BN) \cite{12} which speeds up learning by reducing the problem of internal covariate shifts.

4. Residual neural networks (ResNets) that directly address the problem of vanishing gradients by using skip-layer connections that allow backpropagated error gradients to be mixed with a more direct (linear) error delta from the output layer.
Overall, the basic algorithm and NN model has not changed fundamentally since the 1980’s. Thus, most of the success of deep learning can be ascribed to the availability of big data and faster computers (GPUs) – plus a few clever “hacks”. The training process is expensive, and generally requires a lot of hyperparameter-tuning as well as expert knowledge.

3.1 Related Work

The deep learning “era” was, in part, set off by a related method, namely greedy layer-wise pre-training [7]. This method initializes the NN by bottom-up sequential training of the layers, where each layer is trained by a two-layer autoencoder; i.e. it attempts to learn the identity mapping in each layer. The initialization is followed by running full backprop on the training data. For some smaller datasets, this method has been shown to work well. However, it is no longer used in literature, as the advent of bigger datasets (and other methods) have made it redundant. GoogLeNet [26] is a successful NN architecture that is not trained layer-wise, but which introduces intermediate losses in some of the hidden layers. Deeply supervised nets [16] do apply losses at each hidden layer but have not been equally popular. More recently direct feedback alignment (or random synaptic feedback weights) has been proposed as a more “biologically plausible” method for assigning errors to the hidden layers [18]. Instead of backpropagating exact error gradients through each layer and its non-linearity, random (linear) projections of the output error delta is used. The random feedback weights are fixed during training and the NN learns to adapt to the incoming error signal and achieve error rates similar to those of standard backprop. Difference target-propagation [17] backpropagates target values instead of error gradients. The target value is meant to be close to the activation value while being likely to have provided a smaller loss. With a reasonably good target at hand, a layer-local training criterion can be applied to update each layer separately. Even more recently, Huang et al. [10] showed that you can in fact train the blocks of a ResNet sequentially, one at a time. To do this, they take a boosting [24] approach to block-wise training, and prove that the training error decays exponentially with the depth of the network. Our own method and motivation for layer-wise training is very much in tune with this boosting view.

4 Previous Work

4.1 Speeding Up Distributed Learning

Large-scale distributed learning plays an ever-more increasing role in modern computing. However, whether using a compute cluster with thousands of nodes, or a single multi-GPU machine, the most significant bottleneck is that of communication. In particular, communication between parallel processes. In this work, we explore and compare different approaches to quantizing and encoding the parameters of distributed models. We show that, for gradient descent-based learning algorithms, this can be done - without slowing down the convergence, or hurting the generalization of the model. As a result, the time required for training may be reduced significantly.
The general approach to minimizing communication overhead has been just to reduce the number of parameters that are exchanged, e.g. by having fewer parameters in the first place by sparsifying them, exchanging only some of the parameters, or by sending them less frequently [2, 3]. We take a different approach – rather than impoverishing the number of parameters exchanged, we reduce the communication overhead by reducing the number of bits used to transmit each parameter. We achieve this by quantizing the transmitted values. The entropy [25] of the parameters of the network varies with training epoch and the layer of the network that they represent. To take advantage of this, we vary the number of bits used to quantize any value dynamically, based on the observed entropy of the parameter values in any layer, at any epoch. Additional compression is obtained through Huffman coding [11] of the parameters prior to transmission. The additional overhead of conveying the information about the quantization levels and Huffman code dictionaries, so that the machines receiving these communicated values can decode them, is insignificant compared to the actual number of bits needed to represent the parameters. We are thus able to achieve a compression of nearly an order of magnitude in the communication overhead, for no loss of generalization error in the trained network, as evaluated on a standard classification task.

Figure 2: Generalization accuracy vs. quantization level of weights.

Figure 2 shows the achieved classification accuracy as a function of the bit rate of the weights. When we go beyond 6-7 bits we do not gain anything in the precision of the learned classifier.

4.2 Gradient Boosting

Training a deep neural network is a highly non-convex optimization problem that we usually solve using convex methods. For each extra layer, we add to the network, the problem becomes more non-convex, i.e. more curvature is added to the error surface, making the optimization harder. Yet, it is commonplace to add unnecessary curvature at the output layer even though this does not expand the space of functions that the NN can represent. This curvature is then back-propagated through all the previous layers, causing a detrimental increase in the number of ripples in the error surfaces of especially
<table>
<thead>
<tr>
<th>Output Activation</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sigmoid</td>
<td>1.8</td>
<td>98.5</td>
</tr>
<tr>
<td>Tanh</td>
<td>1.7</td>
<td>95.0</td>
</tr>
<tr>
<td>Linear</td>
<td>1.7</td>
<td>73.5</td>
</tr>
</tbody>
</table>

Table 1: Median results (20 trials) on MNIST for a 392-50-10 NN with ReLUs in the hidden layers; final classification error & no. of epochs needed to converge.

the lower layers, which are already the toughest ones to train. This is done, in part, so that we may perceive the outputs as probabilities, even though they technically are not. In this work, we show that saturating output activation functions, such as the commonly used softmax function, impede learning on a number of standard classification tasks. Moreover, we present results showing that the utility of the softmax does not stem from the normalization, as some have speculated [4, 13]. In fact, the normalization makes things worse. Rather, the advantage is in the exponentiation of error gradients. This exponential gradient boosting is shown to speed up convergence and improve generalization.

Table 1 shows what happened when we first applied this view on real data; the MNIST dataset [15]. Training a simple three-layer NN (fully connected) with ReLUs in the hidden layers, we compared the median results obtained over twenty trials with sigmoid, tanh, and linear output activations. The learning rate was fixed, and carefully tuned for each setting, and neither dropout [8], batch normalization [12], nor weight decay was used. The NN trained for 100 epochs, and the point of convergence is set to be the epoch where the minimum classification error was observed. This experiment was repeated multiple times with other hidden activations, and weight initialization schemes and they all gave the same result: with linear output activations, the rate of convergence is reduced by approximately 25 percent (and some moderate improvements in generalization was observed as well). Note, that the softmax is not included in the table for the simple reason that it performed poorly on this NN configuration.

4.2.1 Exponential & Cubic Boosting

From these MNIST results, one might conclude that using linear outputs is always better than using the softmax, but that is not the case. The reason for this lies in the exponentiation of the outputs. For a moment, stop thinking about the softmax in a probabilistic context, and instead view it as the equivalent of linear outputs, with a mean squared error loss, combined with non-linear boosting of the error deltas, $y - t$. From this perspective, it becomes clear that when we have $y_j = \frac{\exp(x_j)}{\sum \exp(x_i)}$, nothing changes with respect to the one-hot classification, but large errors will be exponentiated. This allows the optimizer to take bigger steps towards a minimum, thus leading to faster convergence. An intuitive interpretation of this would be that when we are confident about an error, we can take an exponentially larger step towards minimizing that error. The idea bears some resemblance to momentum, where we gradually speed things up when the error gradients are
consistent.

If exponentiation of error deltas is good, and saturation is bad, it follows that using an “un-normalized” softmax, so to speak, should yield an improvement. That is, simply use linear outputs, \( y = x \), but compute the error gradients as
\[
\nabla L(y,t) \nabla x = \alpha \exp(y) - t.
\]
Alternatively, we can think of it as an exponential output activation with an incorrect gradient formulation imposed on it, i.e. \( y - t \) (this is in fact how we implemented it). As seen in Figure 3a, this simple change does, in fact, lead to a consistent boost in performance. The result was obtained on CIFAR-10 with a 5-layer CNN; four convolutional layers followed by an affine output layer with linear outputs and exponential gradient boosting (exp-GB), and batch normalization in all layers.

Although we can often ignore large negative outputs that yield large negative error deltas, we cannot ignore all of them. This raises the question whether we may further boost performance by also allowing for the exponentiation of large negative errors. The answer is: yes we can! We can use a simple polynomial, \( y = \alpha x^3 + \beta \); let’s call it pow3-GB. Taking another look at Figure 3a, we see that this does indeed work better; following exactly the same trend as observed with exp-GB, that the error drops significantly faster than with the softmax.

These results generalize to larger datasets and deeper models. Thus, we see consistent results on the ImageNet dataset, and for the task of semantic segmentation (pixel-wise classification) on the PASCAL VOC 2012 data. In the latter case, the training time was reduced from six to four days (compared to using the softmax) when training a model from scratch towards a state-of-the-art result.

4.3 Implementing Neural Networks Efficiently

In this (unpublished) work, we have successfully applied multiple strategies for speeding up deep learning on a single GPU. We take the approach that the optimal configurations
and implementations of CUDA kernels can generally not be known in advance for all possible usage scenarios. Thus, we develop a framework for automated generation and aggressive auto-tuning of kernels for neural networks. Furthermore, we observe that interpreted languages, such as Python and Matlab, are not ideal for launching CUDA kernels; especially not when implemented in an object-oriented manner (as it tends to increase the number of function calls). The reason for this is the significant overhead induced by function and method calls in these languages. Getting high utilization on a GPU requires that the GPU’s scheduler constantly has kernel calls lined up in its pipeline, such that the GPU is never idle. This means that even the tiniest delay between CUDA calls may build up to a considerable amount of idle time when we launch thousands of sequential and dependent kernels – as is the case during NN training. However, using interpreted languages has many practical advantages when we do DL research. Therefore, we have built a system that can automatically generate efficient Matlab code for NN training, where the number of function calls required for a kernel launch is minimal. Our system is significantly faster than popular DL software packages. Thus, we have observed training being two times faster than that of IntelNevana’s Neon, and over six times faster than that of e.g. PyTorch and MXNET.

5 Proposed Work

Training deep networks has long had a reputation for being a bit of a “black art”. One aspect is the historical connection to neuroscience and the brain, which adds a veil of mystery to it. Another is a lack of theory in the field. Albeit the past 2-3 years in particular have seen an increased attention to theory, e.g. [1, 5, 6, 19, 20, 21, 22, 23, 27, 28, 29, 30, 31, 32, 33], we still do not understand exactly why the backpropagation algorithm works so well. In spite of its success, the algorithm is still far from optimal. Using it successfully typically requires significant expert knowledge, time, and resources. How to pick the best architecture and tune the hyperparameters is tricky and not well understood. And training a state-of-the-art model on the ImageNet dataset, requires access to supercomputers, and can take several weeks. Our work aims to address these issues.

5.1 Target Encoding

The past decade has seen a constant increase in the number of layers used in deep neural networks. This has spawned a lot of research on how to deal with the resulting increase in the complexity of the optimization problem and how to build systems that can efficiently handle the massive data-processing involved. Target encoding (TE) is a new and simple method for greedy layer-wise training of deep neural networks. This constructive training scheme allows for the incremental addition of layers, such that the final architecture need not be known in advance. We aim to demonstrate empirically that the proposed method is less reliant on hyper-parameter tuning, and advanced methods such as batch normalization. Moreover, we conjecture that TE may reduce the total training time because upper layers are likely to converge faster than the lower ones. With backprop all layers see the data the same number of times, but with TE we can spend our time where it is most needed, so to speak. Lastly, due to the incremental nature of the method, it
allows for training very deep networks on smaller devices, because we no longer need to keep the entire network in memory; we can also use much larger mini-batch sizes, which significantly improves performance on GPUs. Recently, we trained a 1000-layer FNN on the MNIST dataset – on a laptop – using TE. This would have been impossible to do using standard backprop. Not only, would the FNN not fit in memory on a single GPU, it is also not possible to train such a deep network on a small dataset like MNIST. In fact, fitting an FNN with more than 15-20 layers to that dataset does not appear to be possible with standard backprop. This raises the question: is there a class of functions that can be learned with greedy layer-wise training but not by full backprop?

5.1.1 The Method

Target encoding is similar to the training process for stacked autoencoders (SAEs) [7]; hence the name. Only, instead of using the identity mapping as the target in each layer – we simply use the actual target, $Y$. Thus, we take a dataset, $D(X,Y)$, and a $k$-layer architecture, $\Psi$, and decompose the learning problem into $k-1$ two-layer architectures, $\Psi^{(i)} = (\psi_i, \psi_k)$, and datasets, $D^{(i)} = (Z_i, Y)$, which we train sequentially, one by one, as illustrated in Figure 4.

![Figure 4: Target encoding process for a 4-layer NN. Instead of training all layers of $\Psi$ simultaneously, we train the sub-architectures $\Psi^{(1)}, \Psi^{(2)}$, and $\Psi^{(3)}$ sequentially with $Z_i$ becoming the input to $\Psi^{(i+1)}$.](image)

5.2 Questions to Answer

In our investigation of TE we will attempt to answer the following questions:

1. How does the performance of TE compare to full backprop?

   (a) We hypothesize that, on average, each layer requires fewer epochs to converge, but is that in fact the case?
(b) Do we get the same error rates? Do the non-linearities applied to the error gradients being backpropagated produce better targets for the hidden layers? Or do they just make the optimization problem more non-convex, and thus harder?

(c) Layer-wise training is likely to be more prone to overfitting. Is that actually the case and if so, how can we deal with it?

2. Are hyper-parameters easier to tune for TE?

3. Do we need fewer layers when training DNNs with TE? The incremental nature of the method allows us to stop training when some desired precision is reached.

4. Is there some class of functions which can be learned with greedy layer-wise training, but not by full backprop?

5. Are there any practical / implementational drawbacks of our method? Such as needing the input data to be delivered much faster due to our training shallow networks instead of deep ones?

5.3 Non-Linear Regression by Target Correction

As training DNNs can be viewed as fitting a sequence of non-linear regression models, we now consider the problem of non-linear regression. That is, we wish to fit a function of the form \( f(x) = \sigma(xW) \) to some data \( D(X,Y) \), where \( Y \) is continuous. This problem is commonly solved by gradient-based methods such as stochastic gradient descent. We propose a new method based on the concept of target correction. The idea is quite simple. Imagine that there for a given problem exists a \( W^* \) such that \( Y = \sigma(XW^*) \), then there also exists an intermediate target \( T^* = XX^+T_0 \) that allows us to find \( W^* \) by least squares, i.e. \( W^* = XX^+T^* \). With the target correction method, we first set \( T_0 = Y \), find the least squares solution \( W_0 = XX^+T_0 \), set \( \hat{Y} = \sigma(XW_0) \), and then use the error delta, \( \delta = Y - \hat{Y} \), to make a correction to the target, \( T_1 = T_0 + \eta \delta \). If \( \eta > 0 \) is chosen appropriately, then \( W_1 = XX^+T_1 \) is closer to \( W^* \) than \( W_0 \). For some choice of activation function, \( \sigma \), the iterative target corrections algorithm (ITC) is:

**Algorithm 1 (Iterative Target Corrections).**

**Input:** dataset, \( D(X,Y) \), and learning rate, \( \eta > 0 \)

1. Set \( M := XX^+ \), and \( T := Y \)

2. Repeat until convergence:

   • \( \hat{Y} := \sigma(MT) \)
   • \( T := T + \eta(Y - \hat{Y}) \)

The variable \( \eta \) is equivalent to the learning rate in SGD. One that we may choose to adjust while iterating. For single-bounded \( \sigma \)'s this algorithm can converge in as few as two iterations with the right choice of \( \eta \), while saturating \( \sigma \)'s may require many more; e.g. fifty to one hundred. A possible drawback of ITC is the use of pseudo-inverses, which
can be expensive to compute for large data. However, we need only to compute $X^+$ once, as we can reuse it (see Algorithm 1). Also, it can be computed efficiently in block matrix form, and we can approximate it in a manner similar to SGD by computing it on mini-batches of $X$ and averaging the results. The latter approach allows for data-parallel implementations on distributed systems. We can still use ITC in a NN setting – as long as we have some reasonable targets for the hidden layers. Thus, it is compatible with both backprop and the direct feedback alignment method.

5.3.1 Questions to Answer
In our investigation of ITC we will attempt to answer the following questions:

1. How does ITC perform on non-linear regression problems compared to SGD?
   - Are the error rates better, worse, or the same?
   - Does it converge faster?
   - Is the overall execution time lower for ITC?
   - Is it easier to pick a good learning rate?

2. How can we use ITC to train neural nets?
   - How well does it work with backprop if we use the backpropagated errors as targets for a set of non-linear regression problems?
   - How well does it work with target encoding?

6 Thesis Schedule
6.1 Experiments
For our thesis, we will complete the experiments described below.

6.1.1 Target Encoding
Experiment 1: Simple TE
Train CNNs of varying depths on the MNIST and CIFAR10 datasets using SGD + momentum. Settings:

- Used fixed or no regularization.
- Train each layer for a fixed number of epochs (i.e. no early stopping).
- Use the same learning rate in all the layers.

How do the error and convergence rates compare to full backprop, and is there significant overfitting?

Experiment 2: Regularized TE
Repeat Experiment 1, but apply regularization:
• Use dynamic regularization (apply or increase regularization when signs of overfitting are seen).
• Use early stopping.
• Pick individual learning rates for the layers.
• Try to regularize by training each layer on individual subsets of the data.

Is regularization important, and what are the most effective methods?

**Experiment 3: Windowed TE**
Use the insights gained from Experiments 1 & 2, and apply the best methods to training CNNs on the CIFAR10 data, but this time train windows of hidden layers at a time, and compare the performance to that of training one layer at a time.

• Is there any advantage to training 2-6 layers at a time (in each window)?
• Are the layers “self-regularizing”? I.e. do we need less regularization compared to the single-layer approach?
• Try overlapping windows s.t. the weights of one or more layers that were trained in window $i$ are also being optimized with window $i + 1$.

**Experiment 4: Parallel TE**
Use the insights gained from Experiments 1, 2 & 3, and apply the best methods to training CNNs on the CIFAR10 data. This time, train subsets of the kernels for each layer independently and in parallel. That is, instead of e.g. training 1024 kernels simultaneously on a single worker, train 256 kernels at a time on four workers – without any coordination between the workers. After training, concatenate the representations learned on the four workers and let that be the input to the next layer.

• Try random subsets of the data with balanced classes.
• Try unbalanced classes s.t. some kernels learn more about certain classes than others.
• Try different kernel initialization schemes. E.g. use different principal components for each kernel in order to avoid possible redundancies in the learned kernels.

Is it possible to train NNs in this manner without significantly hurting the performance?

**Experiment 5: What can or cannot be learned by TE**
Address the question: is there a class of functions that can be learned with greedy layer-wise training but not by full backprop, and vice versa?

• Try highly interleaved classes. This problem is known to be hard for full backprop.
• Look to the literature to identify interesting problems, and compare the performance of TE to that of full backprop.
Backprop does not work well, for more than one hidden layer, if $\sigma$ is not monotonic. Does TE do a better job?

**Experiment 6: ImageNet**

Use the insights gained from all the previous experiments, and compare the performance of TE to the state of the art. What is the final conclusion?

### 6.1.2 Target Correction Method

**Experiment 1: Toy data**

Run ITC on toy data:

- Try various activation functions.
- Experiment with different choices for the learning rate.

How does it perform compared to SGD?

**Experiment 2: Bigger data & pseudo-inverses**

Run ITC on real and bigger data:

- Try computing the full pseudo-inverse in one go.
- Try the block matrix pseudo-inverse.
- Try batched and averaged pseudo-inverse.

How does it perform compared to SGD, and what is the most efficient way of computing the pseudo-inverse?

**Experiment 3: ITC & NN training**

How can we use ITC for training NNs?

- Can we use it with standard backprop?
- Can we use it with TE?

How does it perform compared to SGD?

### 6.2 Timeline

1. **By May 20, 2018**: Finish the implementation of target encoding and the required faster method for delivering data during training.

2. **By June 15., 2018**: Compute all relevant baselines on standard datasets.

3. **By July 1, 2018**: Complete all experiments for TE (without ITC).

4. **By August 1, 2018**: Complete all experiments for ITC.

5. **By October 1, 2018**: Complete the writing of the thesis.

Note, that the writing has already commenced and that we will continue writing alongside the execution of our experiments.
6.3 Optional Work

If time permits it, we may attempt further extensions to the greedy layer-wise training. For example, is there some way that we can sufficiently approximate targets for the hidden layers, such that we need not add a decision layer when training each individual layer? If so, this could in fact make the optimization problem convex, for some activation functions, and likely make learning easier and faster.

7 References

References


