Elastic Averaging Stochastic Gradient Descent for Neural networks

Dan Vicente (danvi@kth.se) & **Erik Lindé** (elinde2@kth.se)

April 2024

1 Introduction

Deep learning is a subset of machine learning, which is inspired by the brain and leverages Artificial Neural Network (ANN) architectures with many neuron layers to solve problems within statistical learning. [4] The field has grown in popularity in recent years, in particular due to large language models. One of the most classical problems within machine learning is classification, namely given data

$$\{x_i, t_i\}_{i=1}^N \subset \mathbb{R}^d \times \{0, \dots, K\}$$

for some $K \in \mathbb{N}$, we want to find a mapping $f : \mathbb{R}^d \to \{0, \dots, K\}$, $f(x_i) = y_i$, that minimizes,

$$\mathbb{E}[f(x)], \tag{1}$$

a common loss function in classification is the cross-entropy error, which leads to minimizing the following loss,

$$-\sum_{i=1}^{N} \left[t_i \log(y_i) + (1 - t_i) \log(1 - y_i) \right]. \tag{2}$$

Convolutional Neural Networks (CNNs) are ANNs that make use of convolutional filters to find patterns in data and have shown great prowess with regards to image classification. CNNs rely on various methods of gradient descent for minimizing the loss function 2 [1], of which one method is commonly known as Stochastic Gradient Descent (SGD). SGD is performed by picking a point x_i from the dataset at a time and taking a gradient step to updates the weights of the network,

$$w^{k+1} = w^k - \eta \nabla_w f(x_i, w^k). \tag{3}$$

Here $\eta > 0$ is the learning rate and the gradient with regards to the weight can be computed with the chain rule (backpropagation). Due to the large amount of practical applications of deep learning, the size of many deep networks and the computational intensity of SGD, there is a growing interest in reducing the computational resources by parallelization of the training procedure.

2 Parallelization

In the following project, we will consider a particular method of parallelization of SGD for deep ANNs known as Elastic Averaging Stochastic Gradient Descent (EASGD) [5]. The idea of the EASGD algorithm is to divide the work to be done (computing gradients) into a master process and worker processes. The communication and coordination of the work among the concurrent processes is done cyclically via a so called *elastic force*. This links each process parameter with the master, which stores a center variable.

2.1 Loss function and weight updates in the synchronous case

We assume that we have $P \in \mathbb{N}$ processors and denote by x_t^i, the variable stored by each process at iteration t and $\tilde{x}_t$ the center variable at time t. Then we can formulate the minimization problem over the processes as,

$$\min_{x^1,\dots,x^P,\tilde{x}} \sum_{i=1} \mathbb{E}[f(x^i,\xi^i)] + \frac{\rho}{2}\|x^i - \tilde{x}\|^2, \tag{4}$$

where $g_t^i(x_t^i)$ is denotes the stochastic gradient of $\mathbb{E}[f(x_t^i,\xi^i)]$ and ξ^i follow the distribution $\mathbb{P}$, so that each worker can sample the entire dataset. The EASGD updates are done by taking gradients of the objective function 4. We get the following updates,

$$x_{t+1}^i = x_t^i - \eta(g_t^i(x_t^i) + \rho(x_t^i - \tilde{x}_t) \tag{5}$$

$$\tilde{x}_{t+1} = \tilde{x}_t + \eta \sum_{i=1}^{P} \rho(x_t^i - \tilde{x}_t) \tag{6}$$

If we write $\alpha = \eta\rho$ and $\beta = p\alpha$, then the equations become [5],

$$x_{t+1}^i = x_t^i - \eta g_t^i(x_t^i) - \alpha(x_t^i - \tilde{x}_t) \tag{7}$$

$$\tilde{x}_{t+1} = (1-\beta)\tilde{x}_t + \beta\left(\frac{1}{p}\sum_{i=1}^{p} x_t^i\right) \tag{8}$$

We note that the magnitude of ρ represents the amount of exploration we allow in the model [5], where a small value of ρ allows each x^i to fluctuate further from the center $\tilde{x}$, leading to each process being able to explore more of the loss landscape.

2.2 Loss function and weight updates in the asynchronous case

In order to reduce the added time complexity of process communication, an asynchronous approach to the EASGD algorithm is introduced [5]. Essentially, this approach has fewer communication steps and allows each process to operate more independently. Communication and updates between the master process and workers are only done every few batches. In particular we define a new hyperparameter $\tau > 0$ such that communication between processes and thus, updates of the center variable, are done for each batch such that the iteration τ divides t^i for each process $i = 1, \dots P$.

```
Input: learning rate $\eta$, moving rate $\alpha$,
       communication period $\tau \in \mathbb{N}$
Initialize: $\tilde{x}$ is initialized randomly, $x^i = \tilde{x}$,
            $t^i = 0$
Repeat
    $x \leftarrow x^i$
    if ($\tau$ divides $t^i$) then
        a) $x^i \leftarrow x^i - \alpha(x - \tilde{x})$
        b) $\tilde{x} \leftarrow \tilde{x} + \alpha(x - \tilde{x})$
    end
    $x^i \leftarrow x^i - \eta g^i_{t^i}(x)$
    $t^i \leftarrow t^i + 1$
Until forever
```

Figure 1: Asynchronous EASGD algorithm from [5]

We can denote parameters for the master process as $\tilde{x}$, and for worker process as x^i for an iteration step t^i. If τ does not divide iteration t^i, then x^i is updated as with regular SGD. If τ does divide t^i, then x_i and $\tilde{x}$ are also updated by weighting the value from the previous iteration with value $1 - \alpha$ and the center or worker value respecitvely with α. The algorithm from the paper is illustrated in Figure 1.
Note the role of x, as the updates and gradient should be calculated with this older value rather than the latest available value for stability purposes.

Thus far, we have only implemented asynchronous EASGD. The results and discussion thus focus on this approach. We placed our intial focus on this method as it was not explored as much as the synchronous variant in [5].

2.3 Communication

We now describe the communication used in the asynchronous EASGD. The communication of the worker processes to the master process is done by using MPI in C++. Every τ iteration, workers will communicate with the master process. This involves both sending and receiving model parameters. This communication must be complete before proceeding (at least receiving), as these are then immediately used to update model parameters. The model parameters are sent in an array of floats, for each layer in the network. There are a total of 4866100 parameters for the neural network architecture we use, which we describe in the next section.

For the master process, every τ iteration, it must communicate with each process, with both a send and receive operation. If the communication time would outweigh the computational time for a few batches of SGD for workers, then this could significantly hinder the training of the model. However, as the training time for workers is significant, we do believe that the time workers wait for the master process is fairly small.

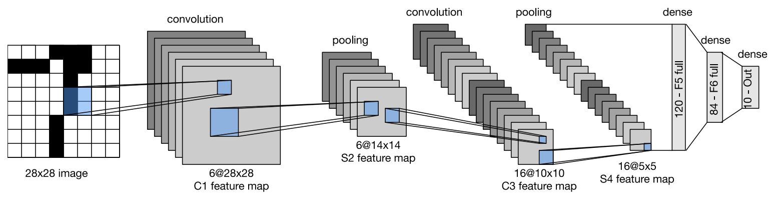

Figure 2: `LeNet-5` [2]

3 Building a Neural Network in C++

There are many frameworks to build and train neural networks, among them are `JAX, TensorFlow, PyTorch`. We have chosen to build our architectures with the C++ API of `PyTorch` [3]. The reason to use libraries and frameworks like these to build neural networks, is that a large number of operations are repeated multiple times and often lead to very complex expressions (gradients and derivatives of multidimensional quantities etc). To build a Convolutional neural network one can consider many different types of architectures. In general, there are two different types of layers in a pure CNN; convolutional layers, which performs kernelized convolutions on subsets of the images and pooling-layers, which aggregate the results from the convolutional layers by performing some operations (commonly, taking the maximum), to learn the important features in each subset of the image. To perform a classification task, one commonly adds a Multilayer Perceptron (MLP) network on the last layer, with K outputs, representing a probability of each class $1, \dots, K$. The class with the highest probability is the classification of the model given an input.

One common class of CNN models is known as `LeNet-5`, which consists of two convolutional layers, two pooling layers and two fully connected MLP layers [2]. A graphical representation of the network can be seen in Figure 2. To build this network in C++ and leverage `pytorch`, we need to make sure that the network inherits all the correct operations and that its parameters are `tensors`. To do this we need to define a class that inherits from the `torch::nn::Module` class [3]. This abstract class inherits all the relevant datastructures that a neural network needs. We need only to specify the network architecture and how the forward pass of the network is computed. The implementation of this can be found in the code snippet below.

```
#include <torch/torch.h>
#include <iostream>
class ConvNetImpl : public torch::nn::Module {
    public:
        explicit ConvNetImpl(int64_t num_classes = 10);
        torch::Tensor forward(torch::Tensor x);
    private:
        torch::nn::Sequential conv1 {
            torch::nn::Conv2d(torch::nn::Conv2dOptions(1, 30, 5).stride(1)),
            torch::nn::ReLU(),
            torch::nn::MaxPool2d(torch::nn::MaxPool2dOptions(2).stride(2))
        };
        torch::nn::Sequential conv2 {
            torch::nn::Conv2d(torch::nn::Conv2dOptions(30, 40, 5).stride(1)),
            torch::nn::ReLU(),
            torch::nn::BatchNorm2d(40),
            torch::nn::MaxPool2d(torch::nn::MaxPool2dOptions(2).stride(2))
        };
        torch::nn::Sequential fc1 {
            torch::nn::Linear(torch::nn::LinearOptions(4*4*40, 700)),
            torch::nn::ReLU()
        };
        torch::nn::Sequential fc2 {
            torch::nn::Linear(torch::nn::LinearOptions(700, 10))
        };
    };
    TORCH_MODULE(ConvNet);
    ConvNetImpl::ConvNetImpl(int64_t num_classes) {
        register_module("conv1", conv1);
        register_module("conv2", conv2);
        register_module("fc1", fc1);
        register_module("fc2", fc2);
    }
    torch::Tensor ConvNetImpl::forward(torch::Tensor x) {
        x = conv1->forward(x);
        x = conv2->forward(x);
        x = x.view({-1, 4*4*40});
        x = fc1->forward(x);
        return fc2->forward(x);
    }
```

When the network architecture has been defined as a `PyTorch` module, we may now implement backpropagation, by defining an optimizer from the `torch::optim` package. Here we have used the SGD module, which has implemented all the relevant methods needed to compute gradients and take gradient descent steps according to the SGD algorithm. Moreover, one needs to specify a loss function in order to compute the gradients and errors. This is done by defining a module from `torch::nn::functional`, here we used `cross_entropy` 2 for the task of classifying the handwritten digits in the `MNIST` dataset. To compare the regular SGD optimizer with the EASGD algorithm, we have added the communication steps to accumulate the center variable, by expanding the tensors containing models parameters and following the steps in Figure 1 while taking into account the communication scheme which we defined in the previous section. Hence, the accumulation of the center variable is done via expanding the tensors into `float` objects, which we can send and recieve with the communication through `MPI`.

Results

We display results for the asynchronous EASGD compared to a traditional SGD in Figure 3. These results were generated on a Macbook Pro from 2021 equipped with the . For a given wall-clock time, we see that the tradtional SGD can offer better training accuracy and results. Although not displayed here, similar results occurred for test data as well. We see that the wall clock time is better with higher τ, which allows for each process to be more independent and has fewer communication steps. As is done in the paper, the parameter α is set according to $\alpha = \frac{\beta}{p\tau}$, which means that increasing τ decreases α. However, by results not presented here, we see that a higher α, for constant number of workers and τ is actually advantageous. Thus the better performance with higher τ is not due to lower α, but actually in spite of a lower α. Similar results are shown for the training loss.

Discussion

The results presented in the previous section are not the desired results. We first note a few sources of error which could impact the results. Firstly, the hardware used can have a significant impact. While running the sequential code, we noted that the sequential code uses 250% of the CPU, whereas when running in parallel, each processes uses only 125%. This means that the parallel process may be disadvantaged, leading to worse results. Also, the MNIST data set might be too elementary for parallelization gains to be evident. The original EASGD paper [5] focused on the CIFAR-10 dataset, which contain images that have more complex spatial patterns. This can be seen in the EASGD papers, where their model spent several hours on training. Our training is orders of magnititude quicker. In fact, the model has a classification accuracy of 99% on the unseen testset after a few minutes of training.

These sources of error will be investigated in the complete analysis. However, their impact may be limited and there can still be important takeaways from the results. Training being slower for lower τ could point to the communication scheme proposed being somewhat ineffective, and we certainly believe there is room for improvement. For example, the communication now sends the model parameters layer by layer, rather than all at once.

Furthermore, there could a fundamental limitation to the asynochronous EASGD used here. [5] did not achieve any remarkable results with asynchronous EASGD or synchronous EASGD, but their significant speedup was instead found with EAMSGD (Elastic Averaging Momentum SGD). We have not yet implemented this algorithm, but will try to implement it for the final draft in hopes of better results.

Acknowledgements

This project is an implementation and extension of [5].

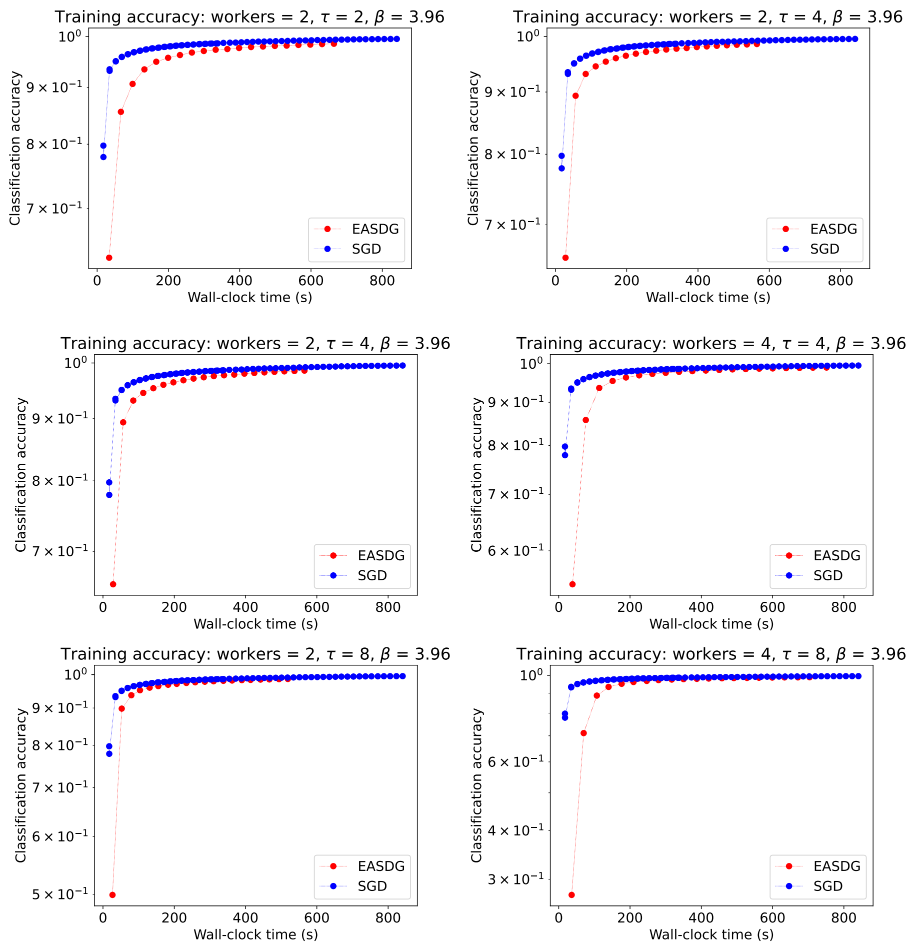

Figure 3: Training accuracy of Asynchronous EASGD vs. normal SGD trainng accuracy.

References

[1] *Gradient Descent*. URL: `https://www.ibm.com/topics/gradient-descent` (visited on 04/25/2024).

[2] *LeNet-5 architecture*. URL: `https://d2l.ai/chapter_convolutional-neural-networks/lenet.html` (visited on 04/25/2024).

[3] Adam Paszke et al. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

[4] *What is deep learning?* URL: `https://www.ibm.com/topics/deep-learning` (visited on 04/25/2024).

[5] Sixin Zhang, Anna Choromanska, and Yann LeCun. *Deep learning with Elastic Averaging SGD*. 2015. arXiv: `1412.6651 [cs.LG]`.