An Overview of Distributed Deep Learning

Seb Arnold

November 23, 2016

1 Introduction

This blog post introduces the fundamentals of distributed deep learning and presents some real-world applications. With the democratization of deep learning methods in the last decade, large - and small ! - companies have invested a lot of efforts into distributing the training procedure of neural networks. Their hope: drastically reduce the time to train large models on even larger datasets. Unfortunately while every commerical product takes advantage of these techniques, it is still difficult for practitioners and researchers to use them in their everyday projects. This article aims to change that by providing a theoretical and practical overview.

Last year, I was lucky to intern at Nervana Systems where I was able to expand their distributed effort. During this 1 year internship, I familiarized myself with a lot of aspects of distributed deep learning and was able to work on topics ranging from implementing efficient GPU-GPU Allreduce routines [1] to replicating Deepind's Gorila [2]. I found this topic so fascinating that I am now researching novel techniques for distributed optimization with Prof. Chunming Wang, and applying them to robotic control [3] with Prof. Francisco Valero-Cuevas.

2 The Problem

2.1 Formulation and Stochastic Gradient Descent

Let's first define the problem that we would like to solve. We are trying to train a neural network to solve a supervised task. This task could be anything from classifying images to playing Atari games or predicting

the next word of a sentence. To do that, we'll rely on an algorithm - and its variants - from the mathematical optimization literature: **stochastic gradient descent**. Stochastic gradient descent (SGD) works by computing the gradient direction of the loss function we are trying to minimize with respect to the current parameters of the model. Once we know the gradient direction - aka the direction of greatest increase - we'll take a step in the opposite direction since we are trying to minimize the final error.

More formally, we can represent our dataset as a distribution χ from which we sample N tuples of inputs and labels $(x_i, y_i) \sim \chi$. Then, given a loss function $\mathcal{L}$ (some common choices include the mean square error, the KL divergence, or the negative log-likelihood) we want to find the optimal set of weights W_{opt} of our deep model F. That is,

$$W_{opt} = \arg\min_{W} \mathbb{E}_{(x,y)\sim\chi}[L(y, F(x; W))]$$

Note: In the above formulation we are not separating the dataset in train, validation, and test sets. However you need to do it !

In this case, SGD will iteratively update the weights W_t at timestep t with $W_{t+1} = W_t - \alpha \cdot \nabla_{W_t} \mathcal{L}(y_i, F(x_i; W_t))$. Here, α is the learning rate and can be interpreted as the size of the step we are taking in the direction of the negative gradient. As we will see later there are algorithms that try to adaptively set the learning rate, but generally speaking it needs to be chosen by the human experimenter.

One important thing to note is that in practice the gradient is evaluated over a set of samples called the minibatch. This is done by averaging the gradient of the loss for each sample in the minibatch. Taking the gradient over the minibatch helps in two aspects.

1. It can be efficiently computed by vectorizing the computations.
2. It allows us to obtain a better approximation of the *true* gradient of $\mathcal{L}(y, F(x; W))$ over χ, and thus makes us converge faster.

However, a very large batch size will simply result in computational overhead since your gradient will not significantly improve. Therefore, it is usual to keep it between 32 and 1024 samples, even when our dataset contains millions of examples.

2.2 Variants of SGD

As we will now see, several variants of the gradient descent algorithm exist. They all try to improve the quality of the gradient by including more or less sophisticated heuristics. For a more in depth treatment, I would recommend Sebastian Ruder's excellent blog post and the CS231n web page on optimization.

2.2.1 Adding Momentum

Momentum techniques simply keep track of a weighted average of previous updates, and apply it to the current one. This is akin to the momentum gained by a ball rolling downhill. In the following formulas, μ is the momentum parameter - how much previous updates we want to include in the current one.

Momentum

$$v_{t+1} = \mu \cdot v_t + \alpha \cdot \nabla \mathcal{L}$$

$$W_{t+1} = W_t - v_{t+1}$$

Nesterov Momentum or Accelerated Gradient [4]

$$v_{t+1} = \mu \cdot (\mu \cdot v + \alpha \cdot \nabla \mathcal{L}) + \alpha \cdot \nabla \mathcal{L}$$

$$W_{t+1} = W_t - v_{t+1}$$

Nesterov's accelerated gradient adds *momentum to the momentum* in an attempt to look ahead for what is coming.

2.2.2 Adaptive Learning Rates

Finding good learning rates can be an expensive process, and a skill often deemed closer to art or dark magic. The following techniques try to alleviate this problem by automatically setting the learning rate, sometimes on a per-parameter basis. The following descriptions are inspired by Nervana's implementation.

Note: In the following formulas, ϵ is a constant to ensure numerical stability, and μ is the decay constant of the algorithm, how fast we decrease the learning rate as we converge.

Adagrad [5]

$$s_{t+1} = s_t + (\nabla\mathcal{L})^2$$
$$W_{t+1} = W_t - \frac{\alpha \cdot \nabla\mathcal{L}}{\sqrt{s_{t+1} + \epsilon}}$$

RMSProp [6]

$$s_{t+1} = \mu \cdot s_t + (1 - \mu) \cdot (\nabla\mathcal{L})^2$$
$$W_{t+1} = W_t - \frac{\alpha \cdot \nabla\mathcal{L}}{\sqrt{s_{t+1} + \epsilon} + \epsilon}$$

Adadelta [7]

$$\lambda_{t+1} = \lambda_t \cdot \mu + (1 - \mu) \cdot (\nabla\mathcal{L})^2$$
$$\Delta W_{t+1} = \nabla\mathcal{L} \cdot \sqrt{\frac{\delta_t + \epsilon}{\lambda_{t+1} + \epsilon}}$$
$$\delta_{t+1} = \delta_t \cdot \mu + (1 - \mu) \cdot (\Delta W_{t+1})^2$$
$$W_{t+1} = W_t - \Delta W_{t+1}$$

Adam [8]

$$m_{t+1} = m_t \cdot \beta_m + (1 - \beta_m) \cdot \nabla\mathcal{L}$$
$$v_{t+1} = v_t \cdot \beta_v + (1 - \beta_v) \cdot (\nabla\mathcal{L})^2$$
$$l_{t+1} = \alpha \cdot \frac{\sqrt{1 - \beta_v^p}}{1 - \beta_m^p}$$
$$W_{t+1} = W_t - l_{t+1} \frac{m_{t+1}}{\sqrt{v_{t+1}} + \epsilon}$$

Where p is the current epoch, that is 1 + the number of passes through the dataset.

2.2.3 Conjugate Gradients

The following method tries to estimate the second order derivative of the loss function. This second order derivative - the Hessian H - is most ably used in Newton's algorithm ($W_{t+1} = W_t - \alpha \cdot H^{-1}\nabla\mathcal{L}$) and gives extremely useful information about the curvature of the loss function. Properly estimating the Hessian (and its inverse) has been a long time challenging task since the Hessian is composed of $|W|^2$ terms. For more information I'd recommend these papers [9–11] and chapter 8.2 of

the deep learning book [12]. The following description was inspired by Wright and Nocedal [13].

$$p_{t+1} = \beta_{t+1} \cdot p_t - \nabla\mathcal{L}$$
$$W_{t+1} = \alpha \cdot p_{t+1}$$

Where β_{t+1} can be computed by the Fletcher-Rieves or Hestenes-Stiefel methods. (Notice the subscript of the gradients.)

Fletcher-Rieves

$$\beta_{t+1} = \frac{\nabla_{W_t}\mathcal{L}^T \cdot \nabla_{W_t}\mathcal{L}}{\nabla_{W_{t-1}}\mathcal{L}^T \cdot \nabla_{W_{t-1}}\mathcal{L}}$$

Hestenes-Stiefel

$$\beta_{t+1} = \frac{\nabla_{W_t}\mathcal{L}^T \cdot (\nabla_{W_t}\mathcal{L} - \nabla_{W_{t-1}}\mathcal{L})}{(\nabla_{W_t}\mathcal{L} - \nabla_{W_{t-1}}\mathcal{L})^T \cdot p_t}$$

3 Beyond Sequentiallity

Let's now delve into the core of this article: distributing deep learning. As mentioned above, when training really deep models on really large datasets we need to add more parallelism to our computations. Distributing linear algebra operations on GPUs is not enough anymore, and researchers have began to explore how to use multiple machines. That's when deep learning met *High-Performance Computing* (HPC).

3.1 Synchronous vs Asynchronous

There are two approaches to parallelize the training of neural networks: model parallel and data parallel. Model parallel consists of "breaking" the learning model, and place those "parts" on different computational nodes. For example, we could put the first half of the layers on one GPU, and the other half on a second one. Or, split layers in their middle and assign them to separate GPUs. While appealing, this approach is rarely used in practice because of the slow communication latency between devices. Since I am not very familiar with model parallelism, I'll focus the rest of the blog post on data parallelism.

Data parallelism is rather intuitive; the data is partitioned across computational devices, and each device holds a copy of the learning

model - called a replica or sometimes worker. Each replica computes gradients on its shard of the data, and the gradients are combined to update the model parameters. Different ways of combining gradients lead to different algorithms and results, so let's have a closer look.

3.2 Synchronous Distributed SGD

In the sychronous setting, all replicas average all of their gradients at every timestep (minibatch). Doing so, we're effectively multiplying the batch size M by the number of replicas R, so that our **overall mini-batch** size is $R \cdot M$. This has several advantages.

1. The computation is completely deterministic.
2. We can work with fairly large models and large batch sizes even on memory-limited GPUs.
3. It's very simple to implement, and easy to debug and analyze.

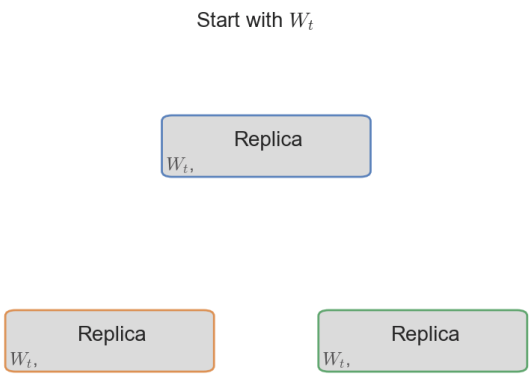

Figure 1:

This path to parallelism puts a strong emphasis on HPC, and the hardware that in use. In fact, it will be challenging to obtain a decent speedup unless you are using industrial hardware. And even so the choice of communication library, reduction algorithm, and other implementation details (e.g., data loading and transformation, model size, ...) will have a strong effect on the kind of performance gain you will encounter.

The following pseudo-code describes synchronous distributed SGD at the replica-level, for R replicas, T timesteps, and M global batch size.

Algorithm 1 Synchronous SGD

while $t < T$ **do**
  Get: a minibatch $(x, y) \sim \chi$ of size M/R.
  Compute: $\nabla\mathcal{L}(y, F(x; W_t))$ on local (x, y).
  AllReduce: sum all $\nabla\mathcal{L}(y, F(x; W_t))$ across replicas into ΔW_t
  Update: $W_{t+1} = W_t - \alpha\frac{\Delta W_t}{R}$
  $t = t + 1$
  (Optional) Synchronize: W_{t+1} to avoid numerical errors

3.3 Asynchronous Distributed SGD

The asynchronous setting is slightly more interesting from a mathematical perspective, and slightly trickier to implement in practice. Each replica will now access a shared-memory space, where the global parameters W_t^G are stored. After copying the parameters in its local memory W_t^L, it will compute the gradients $\nabla\mathcal{L}$ and the update ΔW_t with respect to its current W_t. The final step is to apply ΔW_t^L to the global parameters in shared memory.

Algorithm 2 Asynchronous SGD

while $t < T$ **do**
  Get: a minibatch $(x, y) \sim \chi$ of size M/R.
  Copy: Global W_t^G into local W_t^L.
  Compute: $\nabla\mathcal{L}(y, F(x; W_t^L))$ on (x, y).
  Set: $\Delta W_t^L = \alpha \cdot \nabla\mathcal{L}(y, F(x; W_t^L))$
  Update: $W_{t+1}^G = W_t^G - \Delta W_t^L$
  $t = t + 1$

The advantage of adding asynchrony to our training is that replicas can work at their own pace, without waiting for others to finish computing their gradients. However, this is also where the trickiness resides; we have no guarantee that while one replica is computing the gradients with respect to a set of parameters, the global parameters haven't been updated by another one. If this happens, then the global parameters will be updated with **stale** gradients - gradients computed with old version of the parameters.

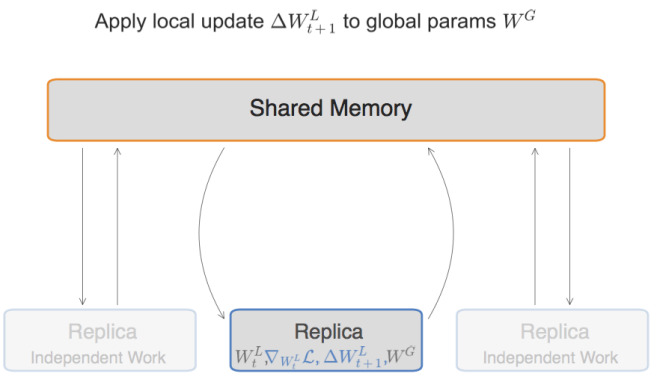

Figure 2:

async nsync special async

3.4 Implementation Tricks

3.5 Hogwild! and Momentum

3.6 Distributed Synthetic Gradients

3.7 Distributed Reinforcement Learning

4 Benchmarks

- toy problems
- mnist
- cifar10

5 A Live Example

6 Acknowledgements

7 Citation

8 References

Some of the relevant literature for this article.

http://www.benfrederickson.com/numerical-optimization/
http://sebastianruder.com/optimizing-gradient-descent/
http://lossfunctions.tumblr.com/
http://www.denizyuret.com/2015/03/alec-radfords-animations-for.html
https://www.allinea.com/blog/201610/deep-learning-episode-3-supercomputer-vs-pong

1. Thakur, R., Rabenseifner, R., Gropp, W.: Optimization of collective communication operations in mpich. International Journal of High Performance Computing Applications. 19, 49–66 (2005).

2. Van Hasselt, H., Guez, A., Silver, D.: Deep reinforcement learning with double q-learning. CoRR, abs/1509.06461. (2015).

3. Arnold, S., Chu, E., Cohn, B., Valero-Cuevas, F.: Performance comparison between trpo and cem. SCMLS. (2016).

4. Nesterov, Y.: A method of solving a convex programming problem with convergence rate o (1/k2). Presented at the.

5. Duchi, J., Hazan, E., Singer, Y.: Adaptive subgradient methods for online learning and stochastic optimization. Journal of Machine Learning Research. 12, 2121–2159 (2011).

6. Hinton, G.: Lecture 6a: Overview of mini-batch gradient descent, (2013).

7. Zeiler, M.D.: ADADELTA: An adaptive learning rate method. arXiv preprint arXiv:1212.5701. (2012).

8. Kingma, D., Ba, J.: Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980. (2014).

9. Dauphin, Y.N., Pascanu, R., Gulcehre, C., Cho, K., Ganguli, S., Bengio, Y.: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In: Advances in neural information processing systems. pp. 2933–2941 (2014).

10. Choromanska, A., Henaff, M., Mathieu, M., Arous, G.B., LeCun, Y.: The loss surfaces of multilayer networks. Presented at the (2015).

11. Martens, J.: Deep learning via hessian-free optimization. In: Proceedings of the 27th international conference on machine learning (icml-10). pp. 735–742 (2010).

12. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning, http://www.deeplearningbook.org, (2016).

13. Wright, S., Nocedal, J.: Numerical optimization.