2.2 Training Deep Neural Networks

The two most important algorithms in deep learning today are gradient descent and backpropagation. Combine, these two algorithms iteratively update the parameters of a neural network until it learns to generalize from the training set.

2.2.1 Gradient Descent

Gradient descent belongs to the class of gradient-based optimization algorithms. The basic idea is to change the parameters until the network produces outputs similar to the training data such that it generalizes to new data. You can imagine a multidimensional landscape where the vertical axis presents the performance of the network and all other dimensions represent the individual weights of the network. As you traverse higher up through the landscape, the performance degrades. As you traverse down the landscape, the performance increases. For the gradient ascent variant, imagine the oppose. The performance is also commonly referred to as cost or loss function. This idea of traversing a cost landscape is the general idea behind all optimization algorithms. The idea behind gradient descent, however, is to always take the steepest path towards a global minimum. This is easy for convex functions (imagine a bowl). In practice, the cost landscape is quite non-convex (imagine lots of mountains, valleys, and plateaus), which can make it difficult to find the right solution. The general form of gradient descent is

\begin{gather\*}

\theta\_{t+1}=\theta\_{t}-\eta\nabla J\_{\theta}(X)

\end{gather\*}

where $\theta$ denotes the network parameters, $\eta$ is the learning rate, and $\nablaJ\_{\theta}$ is the derivative of the cost function $J$ with respect to the parameters $\theta$ and $X$ is a training sample(s).

Batch Gradient Descent

There are several ways to find the steepest gradient. The most basic way of doing this is to compute the gradient for each training sample, average the gradients, and then update the weights with backpropagation. This approach produces very smooth and accurate gradients, but is expensive to compute because it requires the entire training set just to take a single step.

\begin{gather\*}

\theta\_{t+1}=\theta\_{t}-\eta(\frac{1}{N}\sum^{N}\_{i=1}\nabla J\_{\theta}(x\_{i}))

\end{gather\*}

Stochastic Gradient Descent

An alternative approach is to just use a singly randomly sampled training example for each update instead of the entire data set. These updates a noisier, but find solutions much faster and also regularize the network.

\begin{gather\*}

\theta\_{t+1}=\theta\_{t}-\eta\nabla J\_{\theta}(x))

\end{gather\*}

Mini-batch Gradient Descent

A balance between accuracy and time is to use mini-batches. Here, a small batch is of training samples are sampled. The gradient sample is exactly the same at batch gradient descent, averaged over the batch size, $m$, for $n<N$. Mini-batching is so common in the literature that it is also referred to as stochastic gradient descent.

\begin{gather\*}

\theta\_{t+1}=\theta\_{t}-\eta(\frac{1}{N}\sum^{N}\_{i=1}\nabla J\_{\theta}(x\_{i}))

\end{gather\*}

2.2.2 Distributed Gradient Descent

With more compute power, a natural extension is to distribute gradient descent across multiple workers. The first paper demonstrating distributed training was \cite{dean2012large}.

Large-Scale Distributed Deep Networks

In this work, they address GPU memory limitation, limitations of previous approaches to distributed training. First, large models cannot fit in the memory of a single GPU. Typical tricks to address this issue are to divide the computation graph across multiple GPUs on the same machine and share gradients between the GPUs. This transfer can be slow. This limitation also constrains the maximum batch size that can be used. Second, previous approaches attempted to use MapReduce or GraphLab. Both of these methods are ill-equip to handle gradient computations. In that work, the address developing a distributed training framework for deep networks with billions of parameters using large-scale compute clusters with tens of thousands of CPU cores.

Downpour SGD

To scale training to large models, \cite{dean2012large} develop an asynchronous stochastics gradient descent variation call Downpour SGD for training a single neural network. A copy of the model and a partition of the dataset are distributed over several worker nodes. Each epoch, a worker pulls a copy of the current set of parameters from the central parameter server and trains for several iterations updating their local copy of the network. After doing some work, the worker sends theirs new model parameters to the central server to update the global set of parameters and then repeats. Each worker communicates with the parameter server asynchronously and trains locally in parallel. Several consequences of this training paradigm lead to divergence of model parameters and gradients across workers. For instance, each worker could locally train for different number of iterations. Those that train more, can bias the model works their partition of the dataset. Another issue that generally occurs with asynchronous distributed training is stale gradients, which are gradients that correspond to model parameters from an older version of the parameters. To address this, Downpour utilizes the Adagrad optimizer to adaptively update individual weights to cope with staleness. Synchronization and staleness can also be balanced by adjusting the frequency of parameter synchronization and frequency a that gradients are uploaded to the central parameter server. More frequent synchronizations and gradient sharing, decreases staleness and divergence at the cost of more communication. However, the stochasticity introduced from asynchrony provides some regularization.

Sandblaster L-BFGS

To handle batching, the authors introduced the Sandblaster framework for distributed parameter storage and manipulation. Instead of SGD, this framework uses L-BFGS and a coordinator.