Parallel Stochastic Gradient Descent

Olivier Delalleau and Yoshua Bengio
University of Montreal

August 11th, 2007
CIAR Summer School - Toronto
Cost to optimize: $E_z[C(\theta, z)]$ with $\theta$ the parameters and $z$ a training point.

- **Stochastic gradient:**
  
  $$\theta_{t+1} \leftarrow \theta_t - \epsilon_t \frac{\partial C(\theta_t, z_t)}{\partial \theta}$$
Stochastic Gradient Descent

Cost to optimize: \( E_z[C(\theta, z)] \) with \( \theta \) the parameters and \( z \) a training point.

- **Stochastic gradient:**
  
  \[
  \theta_{t+1} \leftarrow \theta_t - \epsilon_t \frac{\partial C(\theta_t, z_t)}{\partial \theta}
  \]

- **Batch gradient:**

  \[
  \theta_{k+1} \leftarrow \theta_k - \epsilon_k \sum_{t=1}^{n} \frac{\partial C(\theta_k, z_t)}{\partial \theta}
  \]
Stochastic Gradient Descent

Cost to optimize: $E_z[C(\theta, z)]$ with $\theta$ the parameters and $z$ a training point.

- **Stochastic gradient:**

  $$\theta_{t+1} \leftarrow \theta_t - \epsilon_t \frac{\partial C(\theta_t, z_t)}{\partial \theta}$$

- **Batch gradient:**

  $$\theta_{k+1} \leftarrow \theta_k - \epsilon_k \sum_{t=1}^{n} \frac{\partial C(\theta_k, z_t)}{\partial \theta}$$

- **Conjugate gradient:** "similar" to batch except the descent direction is not the gradient itself, and the step $\epsilon_k$ is optimized by line search.
Batch is good because:

- Gradient is less noisy (averaged over a large number of samples)
- Can use optimized matrix operations to efficiently compute output and gradient over whole dataset
Batch is good because:
- Gradient is less noisy (averaged over a large number of samples)
- Can use optimized matrix operations to efficiently compute output and gradient over whole dataset

Stochastic is good because:
- For large or infinite datasets, batch and conjugate gradient are impractical.
- More parameters updates ⇒ faster convergence
- Noisy updates can actually help escape local minima
Mini-batch stochastic gradient

"Trade-off" between batch and stochastic:

\[ \theta_{k+1} \leftarrow \theta_k - \epsilon_k \sum_{t=s_k}^{s_k+b} \frac{\partial C(\theta_t, z_t)}{\partial \theta} \]

Typical size of mini-batches: on the order of 10’s or 100’s.
Note: we are using a 16-CPU computer ≠ 100’s computers on a cluster.
Note: we are using a 16-CPU computer ≠ 100’s computers on a cluster.

- Use existing parallel implementations of BLAS to speed-up matrix-matrix multiplications: low speed-up unless using very large mini-batches (⇒ equivalent to batch)
Parallelization

Note: we are using a 16-CPU computer ≠ 100’s computers on a cluster.

- Use existing parallel implementations of BLAS to speed-up matrix-matrix multiplications: low speed-up unless using very large mini-batches (⇒ equivalent to batch)
- Split data into \( c \) chunks (each of the \( c \) CPUs sees one chunk of the data), and perform mini-batch stochastic gradient descent with parameters store in shared memory:

  - Proposed method gives good speed-up in terms of raw "samples/s" speed (e.g.: x13 with 16 CPUs).
Note: we are using a 16-CPU computer ≠ 100’s computers on a cluster.

- Use existing parallel implementations of BLAS to speed-up matrix-matrix multiplications: low speed-up unless using very large mini-batches (⇒ equivalent to batch)
- Split data into c chunks (each of the c CPUs sees one chunk of the data), and perform mini-batch stochastic gradient descent with parameters store in shared memory:
  - if all CPUs are updating parameters simultaneously: poor performance (time wasted waiting for memory write locks)
Parallelization

Note: we are using a 16-CPU computer ≠ 100’s computers on a cluster.

- Use existing parallel implementations of BLAS to speed-up matrix-matrix multiplications: low speed-up unless using very large mini-batches (⇒ equivalent to batch)
- Split data into \( c \) chunks (each of the \( c \) CPUs sees one chunk of the data), and perform mini-batch stochastic gradient descent with parameters store in shared memory:
  - if all CPUs are updating parameters simultaneously: poor performance (time wasted waiting for memory write locks)
  - proposed solution: at any time, only one CPU is allowed to update parameters. The index of the next CPU to update is stored in shared memory, and incremented after each update.

Proposed method gives good speed-up in terms of raw "samples/s" speed (e.g.: x13 with 16 CPUs).
Forking a program creates multiple copies of the program. Memory is duplicated, except for *shared memory*.
Virtual Speed-Up

The speed at which training examples are processed increases (about) linearly with the number of CPUs.

![Graph showing linear relationship between speed-up and number of CPUs]
In practice, the frequency of updates is decreased ⇒ lower speed of convergence.
Suggested experimental setting:
In practice, the frequency of updates is decreased $\Rightarrow$ lower speed of convergence.

Suggested experimental setting:

- Identify target training error $e$ from simple experiments
In practice, the frequency of updates is decreased $\Rightarrow$ lower speed of convergence.

Suggested experimental setting:

- Identify target training error $e$ from simple experiments
- For each number of CPUs $c$, measure the amount of time needed to reach error $e$
In practice, the frequency of updates is decreased $\Rightarrow$ lower speed of convergence.

Suggested experimental setting:

- Identify target training error $e$ from simple experiments
- For each number of CPUs $c$, measure the amount of time needed to reach error $e$

Note that hyper-parameters (learning rate and mini-batch size in particular) need to be re-optimized for each value of $c$. 

Experiment

- Letter recognition dataset from UCI machine learning repository
- 15000 training samples, 16-dimensional input, 26 classes
- Target NLL: 0.11
- Constant network architecture (300 hidden neurons)
- Find optimal fixed learning rate and mini-batch size
Joke

Olivier Delalleau and Yoshua Bengio

Parallel Stochastic Gradient Descent

![Graph showing speed-up versus number of CPUs. The graph indicates an increase in speed-up as the number of CPUs increases, with a peak at 6 CPUs and a decline after that.]
My code is likely buggy!

Funny things can happen in parallel code

Optimization sensitive to noise

⇒ should use different seeds

Should also introduce and optimize a learning rate decrease constant

Use more light-weight parallelization? (pthreads?)

What about large clusters? (MPI)
My code is likely buggy!
My code is likely buggy!

Funny things can happen in parallel code
- My code is likely buggy!
- Funny things can happen in parallel code
- Optimization sensitive to noise ⇒ should use different seeds
My code is likely buggy!

Funny things can happen in parallel code

Optimization sensitive to noise $\Rightarrow$ should use different seeds

Should also introduce and optimize a learning rate decrease constant
My code is likely buggy!

Funny things can happen in parallel code

Optimization sensitive to noise $\Rightarrow$ should use different seeds

Should also introduce and optimize a learning rate decrease constant

Use more light-weight parallelization? (pthreads?)
- My code is likely buggy!
- Funny things can happen in parallel code
- Optimization sensitive to noise $\Rightarrow$ should use different seeds
- Should also introduce and optimize a learning rate decrease constant
- Use more light-weight parallelization? (pthreads?)
- What about large clusters? (MPI)