Systems for Machine Learning

Emmanuel Azuh CSE 550
Outline

- Brief overview of ML and its applications (Broad categories of devices)
- Ecosystem of frameworks for machine learning
- Most famous system for ML on several devices (Tensorflow)
- Support on the two ends of the spectrum (High vs low resource)
ML History

- Two AI winters (determined by funding - eg competition with Japan and pace of technology development to support running the models)
  - 1974 and 1980 slow processing speeds; funding cut from government (till Japan posed competition to become world leader in computer tech)
  - 1987 to 1993 expert systems were bulky and performed poorly so DARPA redirected funding
- Became popularized again when scanning technologies started using CNNs proposed by Yann LeCunn
  - Came back in early 1990s and by early 200s when CNNs processed 10-20% of checks in the US
- Nvidia in 2009 made advances in hardware (GPU) well suited for matrix/vector computations and increased training by about 100x
  - In 2011 for the first time, CNNs outperform humans at visual pattern recognition contest
- Now buzzword for funding (like SDN for networking). Other buzzwords?
Hardware

GPU

~3500 cores vs 16 of Intel Xeon or 32 of Xeon-Phi

FPGA

Highly reprogrammable

TPU

Highly optimized ASIC for inference

[upto 128000 operations per cycle vs tens of thousands in GPU]
Use cases
Tensorflow Overview

DistBelief (2011)

Tensorflow (2015)
Tensorflow Usage Stats

Deep Learning Framework Power Scores 2018

Weights by Category

- KDnuggets Usage Survey: 30%
- GitHub Activity: 10%
- Google Search Volume: 10%
- Medium Articles: 10%
- Amazon Books: 10%
- Online Job Listings: 20%
- ArXiv Articles: 10%

Figure X. Deep Learning Framework Power Scores 2018 by Jeff Hale (Figure by Author)
A toy problem formulation

Linear Regression

\[ Y = WX = [\beta_0 \beta_1][1 \ x]^T \]

Kernel

2-Dimensional Linearly Inseparable Classes

2-Dimensional Linearly Inseparable Classes with Polynomial kernel with Degree 2

\[ x_1^2 + x_2^2 \]
Tensorflow Implementation

import tensorflow as tf

b = tf.Variable(tf.zeros([100]))
W = tf.Variable(tf.random_uniform([784, 100], -1, 1))
x = tf.placeholder(name="x")
relu = tf.nn.relu(tf.matmul(W, x) + b)
C = [...]  # 100-d vector, init to zeroes
            # 784x100 matrix w/rnd vals
            # Placeholder for input
            # ReLU(Wx+b)
            # Cost computed as a function
            # of Relu

s = tf.Session()
for step in xrange(0, 10):
    input = ...construct 100-D input array ...
    result = s.run(C, feed_dict={x: input})  # Create 100-d vector for input
    print step, result
    # Fetch cost, feeding x=input

Figure 1: Example TensorFlow code fragment

Tensors: Multidimensional array (types signed/unsigned int 8 - 64 bits, float, double, complex number)
Create a custom Op

```
#include "tensorflow/core/framework/op.h"
#include "tensorflow/core/framework/shape_inference.h"

using namespace tensorflow;

REGISTER_OP("ZeroOut")
  .Input("to_zero: int32")
  .Output("zeroed: int32")
  .SetShapeFn[](auto c) {
    c->set_output(0, c->input(0));
    return Status::OK();
  });

#include "tensorflow/core/framework/op_kernel.h"

using namespace tensorflow;

class ZeroOutOp : public OpKernel {
public:
  explicit ZeroOutOp(OpKernelConstruction* context) : OpKernel(context) {} 
  
  void Compute(OpKernelContext* context) override {
    // Grab the input tensor
    const Tensor& input_tensor = context->input(0);
    auto input = input_tensor.flat<int32>;

    // Create an output tensor
    Tensor* output_tensor = NULL;
    OP_REQUIRES_OK(context, context->allocate_output(0, input_tensor.shape(),
                                                     &output_tensor));
    auto output_flat = output_tensor->flat<int32>;

    // Set all but the first element of the output tensor to 0.
    const int N = input.size();
    for (int i = 1; i < N, i++) {
      output_flat(i) = 0;
    }

    // Preserve the first input value if possible.
    if (N > 0) output_flat(0) = input(0);
  }
};
```
Device Placement

Device naming: “/job:localhost/device:cpu:0”,"/job:worker/task:17/device:gpu:3"

Figure 3: Single machine and distributed system structure
Device Placement (Grappler)

- **Measurement**
  - Outliers filtering: warmup + robust statistics
  - Captures all the side effects: memory fragmentation, cache pollution, ...
  - Fairly slow and requires access to input data

- **Simulation**
  - Per op cost based on roofline estimates
  - Propagation based on plausible schedule
  - Fast but optimistic and requires robust shape inference

\[ A+2*B+2*C+\text{Identity}(A) \Rightarrow 2*A+2*B+2*C \Rightarrow 2*\text{AddN}(A,B,C) \]

https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf
Device Placement (Reinforcement Learning)

Input
- TensorFlow graph
- Set of available devices

RL model
- Policy
- Evaluate performance

Output
- Assignment of TF graph nodes to devices

Model: NMT with 4 layers
Hardware: 4 K40 GPUS, 1 Haswell CPU
Performance improved by 2.4x

https://web.stanford.edu/class/cs245/slides/TFGraphOptimizationsStanford.pdf
Multi Device Execution

Figure 4: Before & after insertion of Send/Receive nodes
Auto Grad

Figure 5: Gradients computed for graph in Figure 2
Partial Execution & Parallel Loop

Figure 6: Before and after graph transformation for partial execution

```
session.run("c:0", ["f:0"])```

Figure 6: Distributed execution of a while-loop.
Data Parallelism

\[
\begin{bmatrix}
  a_{11} & a_{12} & a_{13} \\
  a_{21} & a_{22} & a_{23} \\
  a_{31} & a_{32} & a_{33}
\end{bmatrix} \times \begin{bmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22} \\
  b_{31} & b_{32}
\end{bmatrix} = \begin{bmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22} \\
  c_{31} & c_{32}
\end{bmatrix}
\]

//Matrix Multiplication
for(i=0; i<row_length_A; i++)
{
    for (k=0; k<column_length_B; k++)
    {
        sum = 0;
        for (j=0; j<column_length_A; j++)
        {
            sum += A[i][j]*B[j][k];
        }
        C[i][k]=sum;
    }
}

#pragma omp parallel for schedule(dynamic,1) collapse(2)
for(i=0; i<row_length_A; i++)
{
    for (k=0; k<column_length_B; k++)
    {
        sum = 0;
        for (j=0; j<column_length_A; j++)
        {
            sum += A[i][j]*B[j][k];
        }
        C[i][k]=sum;
    }
}
Tensorflow Data Parallelism

Synchronous Data Parallelism

Asynchronous Data Parallelism

Figure 7: Synchronous and asynchronous data parallel training
Model Parallel Training & Concurrent Steps

Figure 8: Model parallel training

Figure 9: Concurrent steps
Figure 10: TensorBoard graph visualization of a convolutional neural network model
Distributed Training: Use Case in Uber

Figure 2: The “data parallel” approach to distributed training involves splitting up the data and training on multiple nodes in parallel. In synchronous cases, the gradients for different batches of data are calculated separately on each node but averaged across nodes to apply consistent updates to the model copy in each node.

Bottleneck?

Averages All the Gradients

Parameter Server

Worker A  Worker B  Worker C

Each Averages Portion of the Gradients

Parameter Server A  Parameter Server B  Parameter Server C

Worker A  Worker B  Worker C

All Scatter + All Reduce

https://andrew.gibiansky.com/blog/machine-learning/baidu-allreduce/
Uber’s Horovod

Device Specific Kernel Accelerator: Eyeriss

Fig. 1. Computation of a CNN layer.

Input image

Convolution Kernel

\[
\begin{bmatrix}
-1 & -1 & -1 \\
-1 & 8 & -1 \\
-1 & -1 & -1 \\
\end{bmatrix}
\]

Feature map
Accelerator: Eyeriss

Fig. 2. Eyeriss system architecture.

Fig. 12. PE architecture. The datapaths in red show the data gating logic to skip the processing of zero ifmap data.
Accelerator: Eyeriss
Eyeriss: Run Length Coding

Input: 0, 0, 12, 0, 0, 0, 53, 0, 0, 22, ...

Output (64b):

<table>
<thead>
<tr>
<th>Run Level</th>
<th>Run Level</th>
<th>Run Level</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>12</td>
<td>4</td>
<td>53</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Bar chart showing DRAM Access (MB) with AlexNet Conv Layer.

Uncompressed: 1.2x, 1.4x, 1.7x, 1.8x, 1.9x
Compressed: 6.0x, 5.0x, 4.0x, 3.0x, 2.0x
Tensorflow: Federated Learning on mobile

Tensorflow: Federated Learning on mobile

Figure 3: Actors in the FL Server Architecture

3 Device

Figure 2: Device Architecture