Lecture 3: Overview of Deep Learning System

CSE599W: Spring 2018
The Deep Learning Systems Juggle

We won’t focus on a specific one, but will discuss the common and useful elements of these systems.
We will have lectures on each of the parts!
Typical Deep Learning System Stack

User API

- Programming API
- Gradient Calculation (Differentiation API)
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Example: Logistic Regression

Data

\[ x_i = \begin{bmatrix} \text{pixel}_1 \\ \text{pixel}_2 \\ \vdots \\ \text{pixel}_m \end{bmatrix} \]

Fully Connected Layer

\[ h_k = w_k^T x_i \]

Softmax

\[ P(y_i = k | x_i) = \frac{\exp(h_k)}{\sum_{j=1}^{10} \exp(h_j)} \]
Logistic Regression in Numpy

```python
import numpy as np
from tinyflow.datasets import get_mnist

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x

# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))

    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)

    # update
    W = W - learning_rate * W_grad
```

Forward computation:
Compute probability of each class y given input

- Matrix multiplication
  - np.dot(batch_xs, W)
- Softmax transform the result
  - softmax(np.dot(batch_xs, W))
import numpy as np
from tinyflow.datasets import get_mnist

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x

# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad

Logistic Regression in Numpy

Manually calculate the gradient of weight with respect to the log-likelihood loss.

Exercise: Try to derive the gradient rule by yourself.
import numpy as np
from tinyflow.datasets import get_mnist

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x

# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad

Weight Update via SGD

\[ w \leftarrow w - \eta \nabla_w L(w) \]
Discussion: Numpy based Program

import numpy as np
from tinyflow.datasets import get_mnist

def softmax(x):
    x = x - np.max(x, axis=1, keepdims=True)
    x = np.exp(x)
    x = x / np.sum(x, axis=1, keepdims=True)
    return x

# get the mnist dataset
mnist = get_mnist(flatten=True, onehot=True)
learning_rate = 0.5 / 100
W = np.zeros((784, 10))
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    # forward
    y = softmax(np.dot(batch_xs, W))
    # backward
    y_grad = y - batch_ys
    W_grad = np.dot(batch_xs.T, y_grad)
    # update
    W = W - learning_rate * W_grad

● Talk to your neighbors 2-3 person:

● What do we need to do to support deeper neural networks

● What are the complications
Logistic Regression in Numpy

- **Computation in Tensor Algebra**
  - `softmax(np.dot(batch_xs, W))`

- **Manually calculate the gradient**
  - `y_grad = y - batch_ys`
  - `W_grad = np.dot(batch_xs.T, y_grad)`

- **SGD Update Rule**
  - `W = W - learning_rate * W_grad`
import tinyflow as tf
from tinyflow.datasets import get_mnist

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)

# Training Loop
with tf.Session() as sess:
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
Logistic Regression in TinyFlow

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)

# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Loss function **Declaration**

\[
P(label = k) = y_k
\]

\[
L(y) = \sum_{k=1}^{10} I(label = k) \log(y_i)
\]
Logistic Regression in TinyFlow

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)

# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Automatic Differentiation: Details in next lecture!
Logistic Regression in TinyFlow

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)

# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

**SGD update rule**
Logistic Regression in TinyFlow

```python
import tinyflow as tf
from tinyflow.datasets import get_mnist

# Create the model
x = tf.placeholder(tf.float32, [None, 784])
W = tf.Variable(tf.zeros([784, 10]))
y = tf.nn.softmax(tf.matmul(x, W))

# Define loss and optimizer
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y), reduction_indices=[1]))

# Update rule
learning_rate = 0.5
W_grad = tf.gradients(cross_entropy, [W])[0]
train_step = tf.assign(W, W - learning_rate * W_grad)

# Training Loop
sess = tf.Session()
sess.run(tf.initialize_all_variables())
mnist = get_mnist(flatten=True, onehot=True)
for i in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Real execution happens here!
The Declarative Language: Computation Graph

- Nodes represents the computation (operation)
- Edge represents the data dependency between operations

Computational Graph for \( a \times b + 3 \)
Computational Graph Construction by Step

\[
x = \text{tf.placeholder}(\text{tf.float32}, [\text{None}, 784])
\]
\[
W = \text{tf.Variable}(\text{tf.zeros}([784, 10]))
\]
\[
y = \text{tf.nn.softmax}(\text{tf.matmul}(x, W))
\]
Computational Graph by Steps

\[ y_\_ = tf.placeholder(tf.float32, [\text{None, 10}]) \]

\[ \text{cross}_\text{entropy} = tf.reduce_mean(-tf.reduce_sum(y_\_ \* tf.log(y), \text{reduction_indices}=[1])) \]
Computational Graph Construction by Step

\[ W_{\text{grad}} = \text{tf.gradients(crossentropy, [W])}[0] \]

Automatic Differentiation, detail in next lecture!
Computational Graph Construction by Step

\[
\text{train_step} = \text{tf.assign}(W, W - \text{learning_rate} \times W_{\text{grad}})
\]
Execution only Touches the Needed Subgraph

```
sess.run(train_step, feed_dict={x: batch_xs, y_:batch_ys})
```
Discussion: Computational Graph

- What is the benefit of computational graph?
- How can we deploy the model to mobile devices?
Discussion: Numpy vs TF Program

What is the benefit/drawback of the TF model vs Numpy Model
Typical Deep Learning System Stack

**System Components**

- Programming API
- Gradient Calculation (Differentiation API)
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Computation Graph Optimization

- E.g. Deadcode elimination
- Memory planning and optimization
- What other possible optimization can we do given a computational graph?
Parallel Scheduling

- Code need to run parallel on multiple devices and worker threads
- Detect and schedule parallelizable patterns
- Detail lecture on later

MXNet Example

```python
>>> import mxnet as mx
>>> A = mx.nd.ones((2,2)) * 2
>>> C = A + 2
>>> B = A + 1
>>> D = B * C
```
Typical Deep Learning System Stack

- Programming API
- Gradient Calculation (Differentiation API)
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
GPU Acceleration

- Most existing deep learning programs run on GPUs
- Modern GPU have Teraflops of computing power
Typical Deep Learning System Stack

Not a comprehensive list of elements
The systems are still rapidly evolving :)

**User API**
- Programming API
- Gradient Calculation (Differentiation API)

**System Components**
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling

**Architecture**
- GPU Kernels, Optimizing Device Code
- Accelerators and Hardwares
Supporting More Hardware backends
Each Hardware backend requires a software stack

- Programming API
- Gradient Calculation (Differentiation API)
- Computational Graph Optimization and Execution
- Runtime Parallel Scheduling

- CUDA Library
- MKL Library
- TPU Library
- ARM Library
- JS Library

Hardware
New Trend: Compiler based Approach

Programming API

Gradient Calculation (Differentiation API)

Computational Graph Optimization and Execution

Runtime Parallel Scheduling

High level operator description

Tensor Compiler Stack

Hardware
Links

- TinyFlow: 2K lines of code to build a TensorFlow like API
  - [https://github.com/dlsys-course/tinyflow](https://github.com/dlsys-course/tinyflow)

- The source code used in the slide
  - [https://github.com/dlsys-course/examples/tree/master/lecture3](https://github.com/dlsys-course/examples/tree/master/lecture3)