Chainer & DL4J

CSE 5194.01 – High performance deep learning
Nithin Senthil Kumar
Chainer Overview

- Chainer, a Python based, standalone open source framework for deep learning models.
- First Version Released in 2015 by Preferred Networks
- Development Partnership – Toyota, IBM, Microsoft, Nvidia
- Open source Framework - https://github.com/chainer/chainer
- Key Features:
  - Written purely in Python, built on top of NumPy and CuPy.
  - Define-by-run scheme
  - Performance on large scale systems
- Development stopped as of December 2019 – transitioned to PyTorch
Introduction – DL Frameworks

- Existing deep learning frameworks (2015)
  - Designed mainly for CNNs - Caffe, Torch, PyLearn2
  - Effective for fixed data length
  - Use Domain Specific Languages

- More recently, new types of deep learning models other than CNNs have become major topics of research.
  - Recurrent neural networks (RNNs) - promising results on variable-length data

- Chainer, a versatile open source software framework for deep learning that provides simple and efficient methods for implementing complex algorithms, training models, and tuning model parameters.
DL Frameworks - Approaches

Define-and-Run

- In typical NN frameworks, models are built in two phases - Define-and-Run

- Define phase
  - Instantiation of a neural network object based on a model definition
  - Computational graph is constructed
  - Specifies the inter-layer connections
  - Initial weights
  - Activation functions.
  - The graph is built in memory and the forward computation is set.
  - Backward computation for back propagation - automatic gradient functionalities.

- Run phase
  - Given a set of training examples.
  - Model is trained on a training data set
  - The model is trained by minimizing the loss function using optimization algorithms such as stochastic gradient descent.
Define-and-Run
Define-and-Run – Why?

- Static NN models, such as CNNs, can be easily implemented.
- The model definition may be written in a specific markup language
  - Protobuf for Caffe or YAML for PyLearn2
  - Example – [Sample Model Definition for Caffe](#)
- The deep learning framework then acts as an interpreter
- The NN program does the following
  - receives inputs (data examples)
  - processes these inputs (forward/backward computation)
  - changes the models internal state (updating)
  - outputs the results (predictions).
Caveats of Define-and-Run

• Inefficient Memory Usage
  • Entire Model Must Reside in Memory even if some layers are not needed

• Limited Extensibility
  • In order to maintain backward compatibility
  • Users have two choices: Fork or hack the existing code base
  • Both are suboptimal solutions
    • Forking divides contributions
    • Hacking decreases efficiency of development
    • Creates difficulties in source code maintenance

• Inner workings of neural network are not accessible to user
  • Essentially a black box – Object of a large class
  • Difficulty in debugging
  • Profilers or debuggers cannot be used
Chainer - Design

- Straight forward implementation of more complex deep learning architectures.
- Python – Multi Dimensional array support - NumPy
- Chainer implements CuPy to allow GPU use for faster computations
- Supports popular optimization methods - SGD, AdaGrad, RMSprop, Adam
- Automatic gradients can also be computed for back propagation.
- Many numerical operations for building neural networks, such as convolutions, losses, and activation functions are implemented as Functions.
Define-by-Run

- The most unique aspect of Chainer is the way in which a model’s definition is closely related to its training.

- In contrast to other frameworks, Chainer does not fix a model’s computational graph before the model is trained.

- Instead, the computational graph is implicitly memorized when the forward computation for the training data set takes place.

- Chainer’s approach is unique: building the computational graph “on-the-fly” during training.

- This new approach is called Define-by-Run.

- Define-by-run has gained popularity since the introduction by Chainer and is now implemented in many other frameworks, including PyTorch and TensorFlow.
Define-by-Run

Define-by-Run

- Dynamic computation graphs – variable work

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Source: https://hackernoon.com/how-is-pytorch-different-from-tensorflow-2c90f44747d6
Graph memorization

```python
>>> from chainer import Variable
>>> import numpy as np
```

A class `Variable` represents the unit of computation by wrapping `numpy.ndarray` in it.

```python
>>> x = Variable(np.asarray([[0, 2],[1, -3]]).astype(np.float32))
>>> print(x.data)
[[ 0.   2.]
 [ 1.  -3.]]
```

Users can define operations and functions (instances of `Function`) directly on `Variables`

```python
>>> y = x ** 2 - x + 1
>>> print(y.data)
[[ 1.  3.]
 [ 1. 13.]]
```
Graph memorization

• Since Variables remember what they are generated from, Variable y has the additive operation as its parent (.creator).

```python
>>> print(y.creator)
<chainer.functions.math.basic_math.AddConstant at 0x7f939xxxxx>
```

• Backward computation possible.
  • Tracking back the entire path from the final loss function to the input
  • Input was memorized through the execution of forward computation
  • Computational graph was not defined in advance.

• Many numerical operations and activation functions are given in chainer.functions.
Define by Run

1. Build a computational graph from network definition
2. Input training data and compute the loss function
3. Update the parameters using an optimizer and repeat until convergence
Multi-Layer Perceptron

```python
# (1) Function Set definition
model = FunctionSet(
    11=F.Linear(784, 100),
    12=F.Linear(100, 100),
    13=F.Linear(100, 10)).to_gpu()
# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)
# (3) Forward computation
def forward(x, t):
    h1 = F.relu(model.11(x))
    h2 = F.relu(model.12(h1))
    y = model.13(h2)
    return F.softmax_cross_entropy(y, t)
# (4) Training loop
for epoch in xrange(n_epoch):
    for i in xrange(0, N, b_size):
        x = Variable(to_gpu(...))
        t = Variable(to_gpu(...))
        opt.zero_grads()
        loss = forward(x, t)
        loss.backward()
        opt.update()
```
Key Takeaways

• Network definition is simply represented in Python
• No domain specific language required
• Users can make changes to the network in each iteration (forward computation)
• Standard debugger and profiler can be used
• More complex Neural Networks can be implemented
Recurrent Neural Networks

Visualization

RNN using Chainer

```python
# (1) Function Set definition
model = FunctionSet(
    emb=F.EmbedID(1000, 100),
    x2h=F.Linear(100, 50),
    h2h=F.Linear(50, 50),
    h2y=F.Linear(50, 1000)).to_gpu()

# (2) Optimizer Setup
opt = optimizers.SGD()
opt.setup(model)

# (3) One step forward
def fwdlstep(h, w, t):
    x = F.tanh(model.emb(w))
    h = F.tanh(model.x2h(x) + model.h2h(h))
    y = model.h2y(h2)
    return h, F.softmax_cross_entropy(y, t)

# (4) Full RNN forward computation
def forward(seq)
    h = Variable() # init state
    loss = 0
    for curw, nextw in zip(seq, seq[1:]):
        w = Variable(curw)
        t = Variable(nextw)
        h, new_loss = fwdlstep(h, w, t)
        loss += new_loss
    return loss
```
BENCHMARKING: CHAINER VS CAFFE

### Table 1: Mean time for computations on various networks (msec).

<table>
<thead>
<tr>
<th></th>
<th>Batchsize</th>
<th>conv1</th>
<th>conv2</th>
<th>conv3</th>
<th>conv4</th>
<th>conv5</th>
<th>AlexNet</th>
<th>Overfeat</th>
<th>VGG</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Forward</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chainer 1st</td>
<td>64</td>
<td>245.62</td>
<td>253.08</td>
<td>266.21</td>
<td>199.95</td>
<td>199.80</td>
<td>128</td>
<td>128</td>
<td>16</td>
</tr>
<tr>
<td>Chainer 2-11th</td>
<td>32</td>
<td>72.31</td>
<td>69.00</td>
<td>85.17</td>
<td>29.14</td>
<td>30.68</td>
<td>222.14</td>
<td>406.91</td>
<td>145.77</td>
</tr>
<tr>
<td>Caffe 1-10th</td>
<td>64</td>
<td>47.14</td>
<td>34.66</td>
<td>59.54</td>
<td>25.19</td>
<td>28.48</td>
<td>122.89</td>
<td>368.11</td>
<td>84.54</td>
</tr>
<tr>
<td><strong>Backward</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chainer 1st</td>
<td>103.01</td>
<td>101.14</td>
<td>90.44</td>
<td>39.00</td>
<td>48.39</td>
<td></td>
<td>293.87</td>
<td>553.65</td>
<td>238.88</td>
</tr>
<tr>
<td>Chainer 2-11th</td>
<td>85.48</td>
<td>92.33</td>
<td>82.58</td>
<td>27.89</td>
<td>34.74</td>
<td></td>
<td>277.22</td>
<td>545.51</td>
<td>227.08</td>
</tr>
<tr>
<td>Caffe 1-10th</td>
<td>101.44</td>
<td>105.76</td>
<td>86.46</td>
<td>29.16</td>
<td>31.43</td>
<td></td>
<td>205.05</td>
<td>471.61</td>
<td>191.18</td>
</tr>
</tbody>
</table>
Eclipse Deeplearning4j is the only deep learning programming library written in Java for the Java virtual machine (JVM).

It is a framework with wide support for deep learning algorithms.

Multiple algorithms include distributed parallel versions that integrate with Apache Hadoop and Spark.

Open-source software released under Apache License 2.0

Released 10th Sep 2019 – 1 year ago

Written in Java, CUDA, C, C++

Powered by its own open-source numerical computing library, ND4J.

Works on both CPUs and GPUs

Framework is composable – Combine nets to create deep nets
Deeplearning4j – MultiLayerConfiguration

- Domain Specific Language to configure Deep Neural Networks
- Everything Starts with a MultiLayerConfiguration
- It organizes the layers and their hyperparameters

```java
MultiLayerConfiguration conf = new NeuralNetConfiguration.Builder()
    .weightInit(WeightInit.XAVIER)
    .activation(Activation.RELU)
    .optimizationAlgo(OptimizationAlgorithm.STOCHASTIC_GRADIENT_DESCENT)
    .updater(new Sgd(0.05))
    // ... other hyperparameters
    .list()
    .backprop(true)
    .build();

.layer(0, new DenseLayer.Builder().nIn(784).nOut(250)
    .build());
```
Quickstart with MNIST

```scala
import scala.collection.JavaConversions._
import org.deeplearning4j.datasets.iterator._
import org.deeplearning4j.datasets.iterator.impl._
import org.deeplearning4j.nn.api._
import org.deeplearning4j.nn.multilayer._
import org.deeplearning4j.nn.graph._
import org.deeplearning4j.nn.conf._
import org.deeplearning4j.nn.conf.inputs._
import org.deeplearning4j.nn.conf.layers._
import org.deeplearning4j.nn.conf.weights._
import org.deeplearning4j.optimize.listeners._
import org.deeplearning4j.datasets.datavec.RecordReaderMultiDataSetIterator
import org.nd4j.evaluation.classification._

import org.nd4j.linalg.learning.config._ // for different updaters like Adam,
import org.nd4j.linalg.activations.Activation // defines different activation
import org.nd4j.linalg.lossfunctions.LossFunctions // mean squared error, mult
```
Build the neural network

```scala
val outputNum = EmnistDataSetIterator.numLabels(emnistSet) // total output classes
val rngSeed = 123 // integer for reproducibility of a random number generator
val numRows = 28 // number of "pixel rows" in an mnist digit
val numColumns = 28

val conf = new NeuralNetConfiguration.Builder()
  .seed(rngSeed)
  .updater(new Adam())
  .l2(1e-4)
  .list()
  .layer(new DenseLayer.Builder()
    .nIn(numRows * numColumns) // Number of input datapoints.
    .nOut(1000) // Number of output datapoints.
    .activation(Activation.RELU) // Activation function.
    .weightInit(WeightInit.XAVIER) // Weight initialization.
    .build()
  )
  .layer(new OutputLayer.Builder(LossFunctions.LossFunction.NEGATIVELOGLikelhood)
    .nIn(1000)
    .nOut(outputNum)
    .activation(Activation.SOFTMAX)
    .weightInit(WeightInit.XAVIER)
    .build()
  )
  .build()
```
Train the Model

```java
// create the MLN
val network = new MultiLayerNetwork(conf)
network.init()

// pass a training listener that reports score every 10 iterations
val eachIterations = 10
network.addListeners(new ScoreIterationListener(eachIterations))

// fit a dataset for a single epoch
// network.fit(emnistTrain)

// fit for multiple epochs
// val numEpochs = 2
// network.fit(emnistTrain, numEpochs)
```
Evaluate the model

```scala
// evaluate basic performance
val eval = network.evaluate[Evaluation](emnistTest)
println(eval.accuracy())
println(eval.precision())
println(eval.recall())

// evaluate ROC and calculate the Area Under Curve
val roc = network.evaluateROCMultiClass[ROCMultiClass](emnistTest, θ)
roc.calculateAUC(classIndex)

// optionally, you can print all stats from the evaluations
print(eval.stats())
print(roc.stats())
```
More Info at:
https://deeplearning4j.konduit.ai/
Thank You!