ONNX

ONNX MEETS FLINK

The long trudge towards integrating PyTorch, Chainer, CNTK, MXNet and other models in Flink streaming applications.

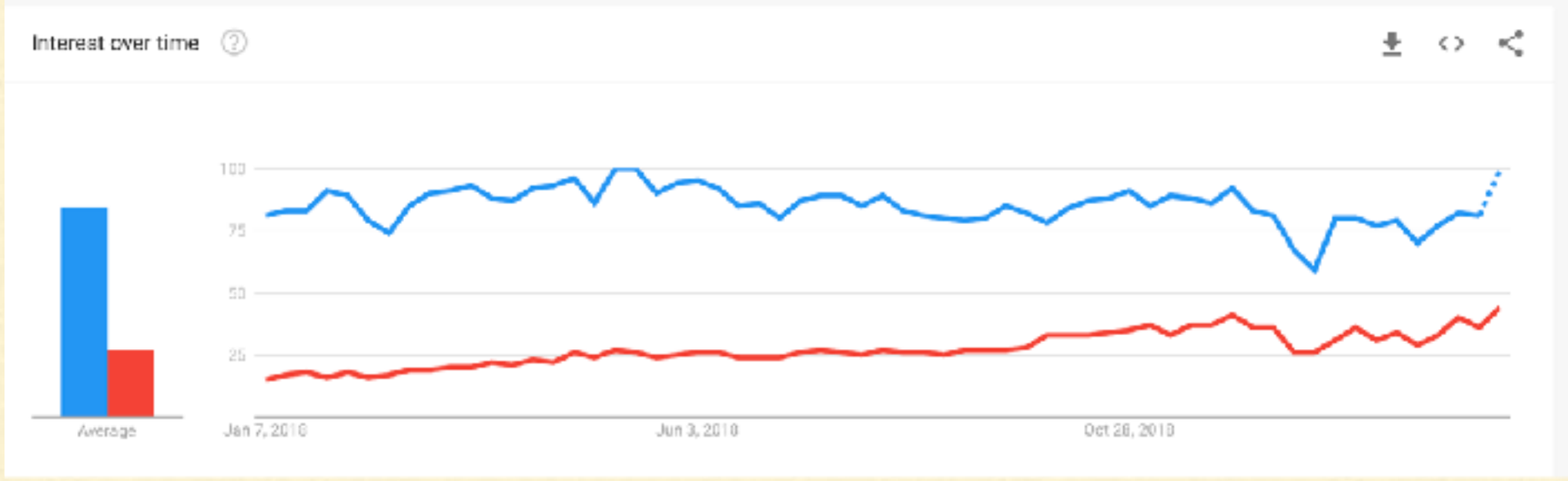
Overview

- The Problem/Motivation
- ONNX
  - Overview
  - Limitations
- End-to-end example with Java Embedded Python

Goals and challenges

- Goals
  - Remove barrier between A.I. “research” and “production.”
  - Enable access to recent state of the art models from major conference and Python based frameworks
    - Specifically, integrate deep learning models written in Python frameworks like PyTorch, CNTK, Chainer into Flink pipelines for realtime inference on streaming data.
- Challenge(s)
  - Poor Python support in Flink and vice-versa poor ONNX support in Java
  - Converting a model to ONNX itself can be quite arduous
  - It can be challenging to rewrite pre-processing code in Java

The rise of PyTorch



- International Conference Learning Representations (ICLR) statistics
  - 2018: 87 papers mentioned PyTorch (compared to 228 that mentioned Tensorflow)
  - 2019 252 papers mentioned PyTorch (compared to 266 that mentioned Tensorflow) Roughly a 190% increase!

PyTorch Powered frameworks

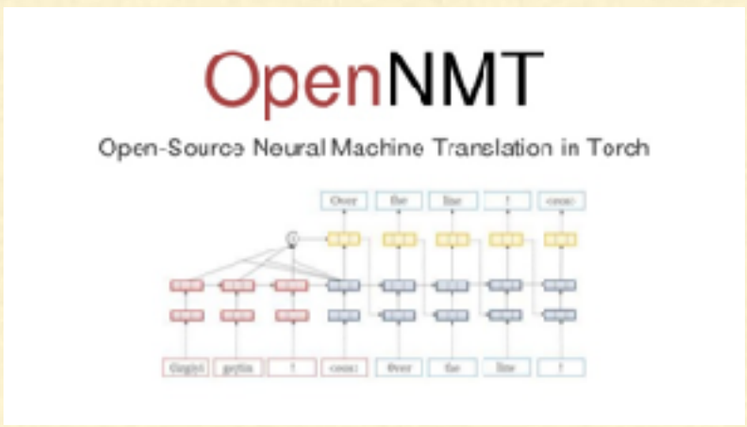
AllenNLP

flair

fast.ai

NLP

ParlAI



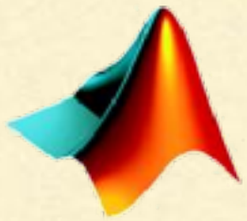
Other

torchvision

torchcv

What is ONNX?

Open neural network exchange format



PaddlePaddle

WHY USE ONNX?

- Backends that (at least in theory) run in a large number of environments.
- Can export models from a variety of formats to a standard format
- Exported models generally smaller (in terms of space) than full models.

Overview of possible ways to integrate ONNX models into Flink

- Create a micro-service and use in conjunction with Flink AsyncIO.

- Use Java embedded Python (JEP) and run using Caffe2 (or Tensorflow)

- Load model natively into Java/Scala and run with a JVM backend framework

ONNX frameworks overview

ONNX Scoreboard Measure supported operations

Framework / tool	Installation	Exporting to ONNX (frontend)	Importing ONNX models (backend)

Latest Update	2018-08-30 19:13:19.606024
ONNX	1.2.2
PyTorch/Caffe2	0.5.0a0+e65f3fc
TensorFlow	1.10.1

Ops

Op	tensorflow	caffe2
ATen (Experimental)	Failed!	Failed!
Abs	Passed!	Passed!
Acos	Passed!	Passed!
Add	Passed!	Passed!
Affine (Experimental)	Failed!	Failed!
And	Passed!	Passed!
ArgMax	Passed!	Passed!
ArgMin	Passed!	Passed!
Asin	Passed!	Passed!
Atan	Passed!	Passed!
AveragePool	Passed!	Passed!
BatchNormalization	Passed!	Passed!
Cast	Passed!	Failed!
Ceil	Passed!	Passed!
Clip	Passed!	Passed!

Models

Model	tensorflow	caffe2
resnet50	9/9 nodes covered: Passed!	9/9 nodes covered: Passed!
bvlc_alexnet	8/8 nodes covered: Passed!	8/8 nodes covered: Passed!
inception_v2	12/12 nodes covered: Passed!	12/12 nodes covered: Passed!
squeezenet_old	7/7 nodes covered: Passed!	7/7 nodes covered: Passed!
inception_v1	10/10 nodes covered: Passed!	10/10 nodes covered: Passed!
vgg19	7/7 nodes covered: Passed!	7/7 nodes covered: Passed!
shufflenet	11/11 nodes covered: Passed!	11/11 nodes covered: Passed!
densenet121	10/10 nodes covered: Passed!	10/10 nodes covered: Passed!
Summary	8/8 model tests passed	8/8 model tests passed

OPTIONS

- ONNX options
  - Current backends
    - Caffe2 (Python, C++)
    - CNTK (C++, C#, Python, **Java experimental)**
    - Tensorflow-ONNX (Python) [Not analogous to Tensorflow)
    - VESPA (**Java**)
    - Menoh (C++, **Java,** C#, Ruby, NodeJS)

Menoh in Java

Only 19 of the 116 ops available (so pretty limited for now)

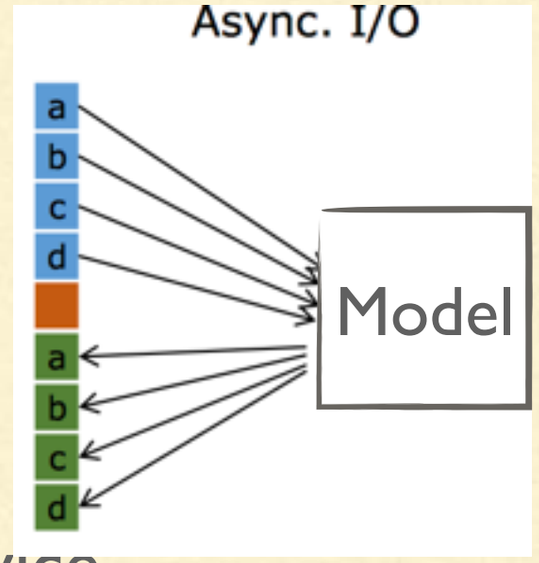
```
import jp.preferred.menoh.ModelRunner;
import jp.preferred.menoh.ModelRunnerBuilder;
try (
    ModelRunnerBuilder builder = ModelRunner
        // Load ONNX model data
        .fromOnnxFile(“squeezenet.onnx”)
        // Define input profile (name, dtype, dims) and output profile (name, dtype)
        // Menoh calculates dims of outputs automatically at build time
        .addInputProfile(conv11InName, DType.FLOAT, new int[] {batchSize, channelNum, height
width})
        .addOutputProfile(fc6OutName, DType.FLOAT)
        .addOutputProfile(softmaxOutName, DType.FLOAT)
       // Configure backend
        .backendName("mkldnn")
        .backendConfig("");
    ModelRunner runner = builder.build()
) {
    // The builder can be deleted explicitly after building a model runner
    builder.close();
```

WHEN NOT TO USE ONNX?

- Export process in many cases is difficult and time consuming!
- Backends have limited support for various operations.
  - For instance, Yolo2 still cannot be run on even Caffe2 or Tensorflow backend due to lack of support of ImageScaler.
- Some models have to be re-trained before exporting

AsyncIO and Microservice

- Flink calls model "API" using AsyncIO similar to any other API connection
- Pros
  - Use Docker container to capture exact model dependencies (smaller container than with Flink+Model)
  - No (extensive) re-writing of code needed
- Cons
  - Have to handle scaling/maintaining a separate service



Java Embedded Python (JEP)

- Uses JNI and the Cython API to start up the Python interpreter inside the JVM
- Faster than many alternatives
- Can use pretty much any Python library including numpy, Tensorflow, PyTorch, Keras, etc
- Automatically converts Java primitives, Strings, and jep.NDArrays sent into the Python interpreter into Python primitives, strings, and numpy.ndarrays

Using PyTorch directly with JEP

- Setup can be a bit painful
  - Have to get Python dependencies on all Flink nodes
  - Job needs path to Python
  - "Unsatisfied Link Error" is very common

Bootstrap script possible for EMR on AWS

Easiest way solution use: Kubernetes

AIStream JEP Flink Docker container

flair

- NLP framework written in PyTorch with a state of the art named entity recognition (NER) model.

```
from flair.data import Sentence
from flair.models import SequenceTagger

# make a sentence
sentence = Sentence('I love Berlin .')

# load the NER tagger
tagger = SequenceTagger.load('ner')

# run NER over sentence
tagger.predict(sentence)
```

Task	Language	Dataset	Flair	Previous best
Named Entity Recognition	English	Conll-03	**93.18** (F1)	*92.22 (Peters et al., 2018)*
Named Entity Recognition	English	Ontonotes	**89.3** (F1)	*86.28 (Chiu et al., 2016)*
Emerging Entity Detection	English	WNUT-17	**49.49** (F1)	*45.55 (Aguilar et al., 2018)*

Task	Language	Dataset	Flair	Previous best
Named Entity Recognition	English	Conll-03	**93.18** (F1)	*92.22 (Peters et al., 2018)*
Named Entity Recognition	English	Ontonotes	**89.3** (F1)	*86.28 (Chiu et al., 2016)*
Emerging Entity Detection	English	WNUT-17	**49.49** (F1)	*45.55 (Aguilar et al., 2018)*

- Easy to train and combine with new methods
- Framework handles complex preprocessing and models PyTorch subclasses (therefore exporting to ONNX is not fun)

Named entity recognition on Flink data stream with Flair

```
import jep.Jep;
import jep.JepConfig;
import org.apache.flink.api.common.functions.RichMapFunction;
import jep.SharedInterpreter;
import org.apache.flink.configuration.Configuration;

public class flairMap extends RichMapFunction<TweetData, String>{
    String tweetText = tweet.tweetText;
    tweetText = tweetText.replaceAll("[^A-Za-z0-9]", " ");
    try {
    @Override
    public void open(Configuration c)
    {    j = new jep.SharedInterpreter();
        try {
         Object result = j.getValue("s.get_spans('ner')");
         return result.toString();
    }
    catch(jep.JepException e){
        e.printStackTrace();
        catch (jep.JepException e) {
            throw e;
            e.printStackTrace();
        }
    }
}
}
public String map(TweetData tweet) throws jep.JepException{
private SharedInterpreter j;
    j.set("text", tweetText);
    j.eval("s=Sentence(text)");
    j.eval("model.predict(s)");
    j.eval("from flair.data import Sentence");
    j.eval("from flair.models import SequenceTagger");
    j.eval("model = SequenceTagger.load('ner')");
```

Sentiment Analysis with Flair

```
from flair.models import TextClassifier
from flair.data import Sentence
classifier = TextClassifier.load('en-sentiment')
sentence = Sentence('Twitter is a really good company!')
classifier.predict(sentence)
# print sentence with predicted labels
print('Sentence sentiment is: ' + sentence.labels)
```

```
public String tweetText(String tweet) throws JepException{
    String tweetText = tweet;
    tweetText = tweetText.replaceAll("[^A-Za-z0-9]", " ");
    try {
        j.set("text", tweetText);
        j.eval("s=Sentence(text)");
        j.eval("model.predict(s)");
        Object result = j.getValue("s.labels");
        return result.toString();
    }
    catch(jep.JepException e){
        e.printStackTrace();
        throw e;

    }
}
```

Putting it all together

https://github.com/isaacmg/dl_java_stream

- Consume data from Twitter Source using Flink Twitter Connector
- Filter out non-English Tweets
  - Alternatively could load multilingual NER model(s)
- Named Entity Recognition on Tweets (remove non-entities)
- Sentiment Analysis on Tweet (entity, label, sentiment)
- Convert to Table. Run query

```
SELECT entity, sentiment, count(entity)
FROM Tweets
GROUP BY entity, sentiment
```

Conclusions

- Currently easiest to either use JEP or a micro-service + AsyncIO
  - Saves time converting model to ONNX
  - No need to re-write code
- Promising frameworks in the works like Menoh, VESPA, DI4J etc should eventually support ONNX natively but aren't mature enough yet.