Distributed C++-Python embedding for fast predictions and fast prototyping

Second Workshop on Distributed Infrastructures for Deep Learning (DIDL18)
Rennes France, December 10, 2018

Georgios Varisteas, SnT, georgios.varisteas@uni.lu
Tigran Avanesov, OLA Mobile, tigran.avanesov@olamobile.com
Radu State, SnT, radu.state@uni.lu
Banner Display Advertising at OLA Mobile

1. User clicks on *generic banner ad*

2. Ad request sent with user profile
   *device, OS, provider, browser, date, country code, etc.*

3. Feature extraction from user profile
   Time localization, Country Code Conversion $\rightarrow$ OneHotEncoding

4. Predict most suitable ad campaign and landing page

5. Redirect user to landing page
Banner Display Advertising at OLA Mobile

1. User clicks on *generic banner ad*

2. Ad request sent with user profile
   
   - *device, OS, provider, browser, date, country code, etc.*

3. Feature extraction from user profile
   
   - Time localization, Country Code Conversion → OneHotEncoding

4. Predict most suitable ad campaign and landing page

5. Redirect user to landing page
Ad Campaign classification

- 1 year of research using SciKit Learn, started in 2016
  - Uses Logistic Regression classifiers
  - Extensive tuning based on sklearn implementation specifics
- Extensive feature engineering
  - Dirty, unstructured, frequently changing data
  - Data patterns do not imply correlations
- Classification based on a boolean label: Sale, not Sale
  - Highly unbalanced datasets:
    - Some got 1 sale every 1M clicks
    - Some got 1 sale every 1K clicks
    - Achieved consistent accuracy above 90%
Online Continuous Training Service

- **ConsumerManager**: Processes live streamed data
  - each file a complete data-set
- **DataManager**: Persists records into structured files
- **TrainingManager**: Train LogisticRegression models
  - one model per predicted feature value
- **Predictor**: Predict sale probability per campaign
  - the predicted feature is the campaign id
  - input is the user profile
Predictor

- Python based implementation
  - REST server
  - Loads classifiers from disk
  - Per request, predicts with every available classifier

- 5000 active campaigns on average, thus 5000 classifiers!
- The 200ms deadline still applies
Predictor

• Python based implementation
  • REST server
  • Loads classifiers from disk
  • Per request, predicts with every available classifier

• 5000 active campaigns on average, thus 5000 classifiers!
  • The 200ms deadline still applies

• Best result after heavy optimization: 3.1ms per classifier
Predictor

• Python based implementation
  • REST server
  • Loads classifiers from disk
  • Per request, predicts with every available classifier

• 5000 active campaigns on average, thus 5000 classifiers!
  • The 200ms deadline still applies

• Best result after heavy optimization: 3.1ms per classifier
  • For 5000 classifiers that would be: **15.5 sec**
Predictor

- Python based implementation
  - REST server
  - Loads classifiers from disk
  - Per request, predicts with every available classifier

- 5000 active campaigns on average, thus 5000 classifiers!
  - The 200ms deadline still applies

- Best result after heavy optimization: 3.1ms per classifier
  - For 5000 classifiers that would be: **15.5 sec**
  - To obey deadline we need **77.5** hardware threads and no conflicts
Python version performance

![Graph showing time vs. classifiers for Python SingleThreaded]
Python version performance

![Graph comparing Python MultiThreading and Python SingleThreaded performance](attachment:graph.png)
Python version performance

• Python GIL: Concurrency does not translate to parallelism
C++ to the rescue

• Embed Python into C++
  • Have C++ handle all data intensive processing
    • Faster and fine grained memory management
    • Lighter data structures
    • Much faster computation execution
    • True massive parallelism
  • Execute predictions in Python
    • 0.8ms per classifier
    • Implement Python multi-processing: no GIL
  • No need to change ML framework
  • No need to change the trainer
C++ to the rescue
C++-Python Performance

![Graph showing performance comparison between C++-Python Embedding, Python MultiThreading, and Python SingleThreaded over classifiers.](image_url)
C++-Python Scalability, 2000 classifiers
C++-Python Worker Utilization
C++-Python Worker Utilization
Deployment statistics
Deployment statistics

October 2018 results: revenue increase by 31%
Conclusions

- Python is dead slow!
- Concurrency does not guarantee performance

- Embedding Python into C++ enabled major performance improvements
- Development time was short
  - No need to evaluate a new ML platform

- Future work
  - improve work distribution, remove central queue
  - improve interprocess shared memory performance