AIM375-P

Natural language modeling with Amazon SageMaker BlazingText algorithm

Denis V. Batalov, PhD
WW Tech Leader, AI/ML
Amazon Web Services

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

Natural Language

Machine Learning

Georgia



azon

geMaker

zingText

The modern Georgian alphabet consists of 33 letters:

ა ani	ბ bani	გ gani	დ doni	ე eni	ვ vini	ზ zeni	თ tani	ი ini	კ k'ani	ლ lasi
მ mani	ნ nari	ო oni	პ p'ari	ჟ zhani	რ rae	ს sani	ტ t'ari	უ uni	ფ pari	ქ kani
ღ ghani	ყ q'ari	შ shini	ჩ chini	ც tsani	ძ dzili	წ ts'ili	ჭ ch'ari	ხ khani	ჯ jani	ჰ hae

Prerequisites

- Fundamentals of ML or DL
- Understanding of NLP or IR problems
- Some familiarity with AWS services

Good-to-have’s

- Knowledge of Python and NumPy
- Knowledge of Jupyter environment

Agenda

- Word Embedding with Word2Vec
- Overview of Amazon SageMaker
- Overview of BlazingText algorithm
- Demo

Word Embedding

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

Word Representations

Word as a Number: Label encoding

“to be or not to be”

Vocabulary: [to=0, be=1, or=2, not=3]

“to be or not to be” → 0, 1, 2, 3, 0, 1

Word as a Vector: 1-hot encoding

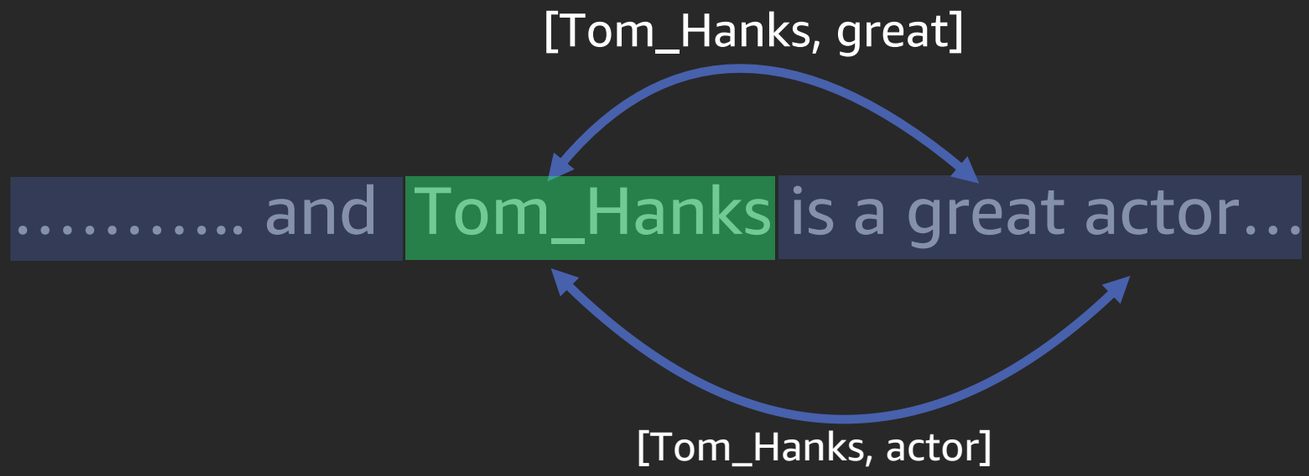
	to	be	or	not	to	be
to	1	0	0	0	1	0
be	0	1	0	0	0	1
or	0	0	1	0	0	0
not	0	0	0	1	0	0

→

2
2
1
1

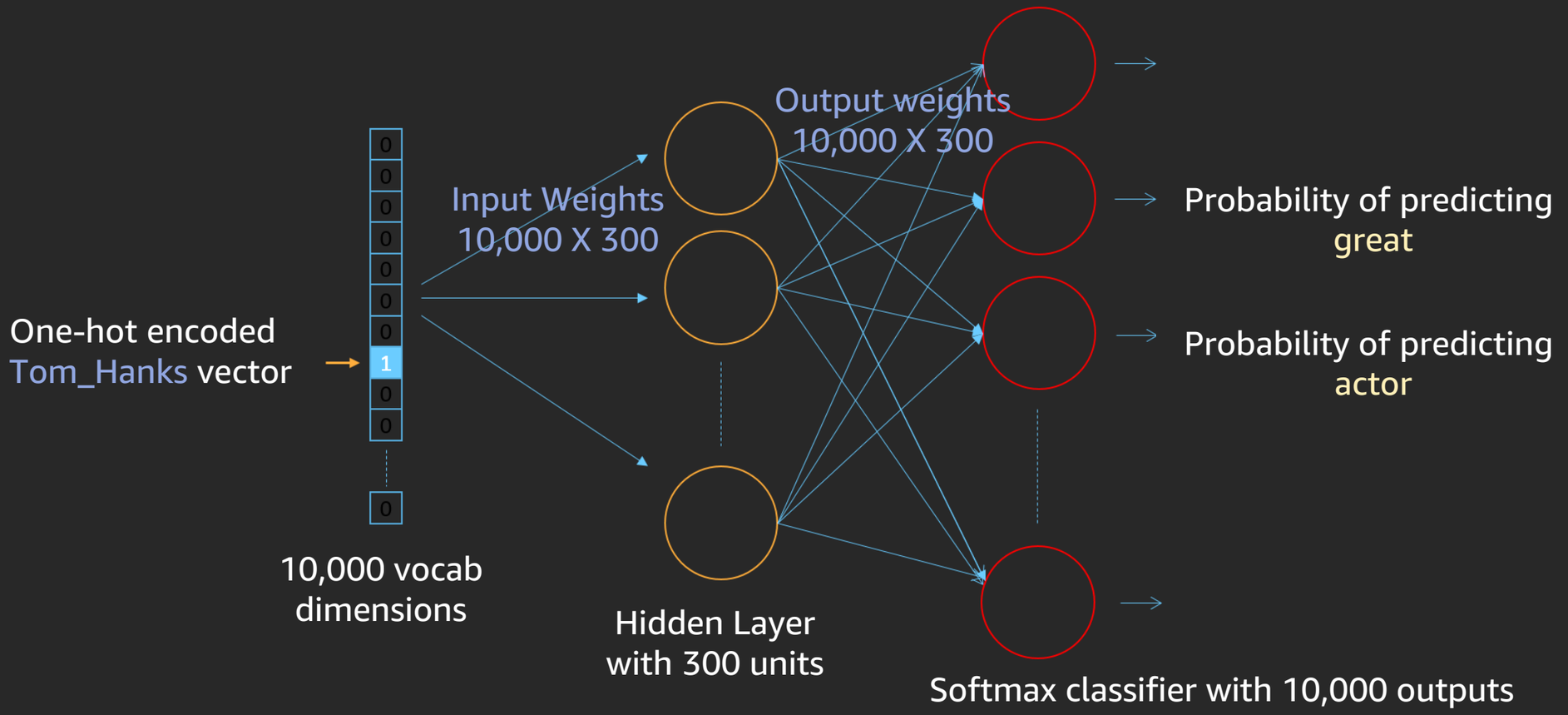
Intuition

Given a sentence - Try to maximize the probability of predicting context words.



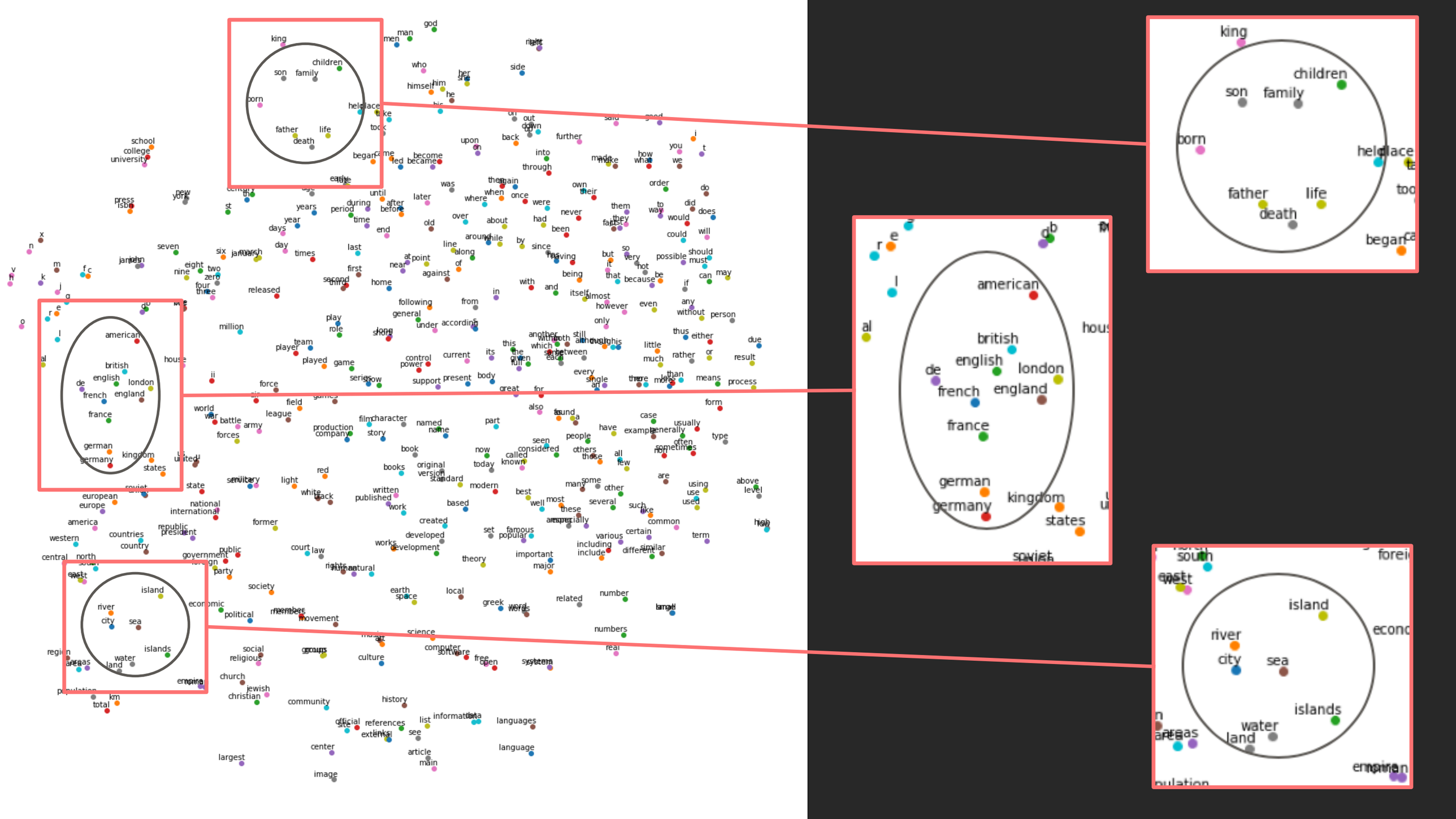
Intuition

Given a sentence - Try to maximize the probability of predicting context words.



Word2Vec Algorithm

- 2013 paper by Mikolov et al., “Efficient Estimation of Word Representations in Vector Space,” gives rise to Word2Vec algorithm
- How to train? Predict surrounding words in a window of length *m* of every word
- Dense, low-dimensional embedding space such that the geometry of resulting vectors captures word semantic and syntactic similarities
- This idea has enabled many Natural Language Processing (NLP) algorithms to achieve better performance – machine translation, POS tagging, NER, Sentiment analysis
- vec(“King”) – vec(“Man”) + vec(”Woman”) = vec(“Queen”)


king
children
son
family
born
father
life
death
american
british
english
london
france
england
german
germany
kingdom
states
island
river
city
sea
islands
water
land

Overview of Amazon SageMaker

AWS re:Invent

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

aws

THE AWS ML & AI STACK

Broadest and deepest set of capabilities

AI Services

VISION	SPEECH	LANGUAGE	CHATBOTS	FORECASTING	RECOMMENDATIONS
AMAZON REKOGNITION IMAGE AMAZON REKOGNITION VIDEO AMAZON TEXTRACT	AMAZON POLLY AMAZON TRANSCRIBE	AMAZON TRANSLATE AMAZON COMPREHEND &COMPREHEND MEDICAL	AMAZON LEX	AMAZON FORECAST	AMAZON PERSONALIZE

ML Services

Amazon SageMaker

Ground Truth	Notebooks	Algorithms + Marketplace	Reinforcement Learning	Training	Optimization	Deployment	Hosting

ML Frameworks + Infrastructure

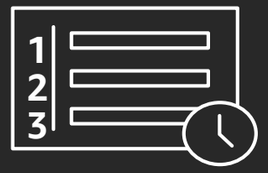
FRAMEWORKS	INTERFACES
TensorFlow, mxnet, PYTORCH	GLUON, Keras

INFRASTRUCTURE
EC2 P3 & P3DN, EC2 G4, EC2 C5, FPGAS, DL CONTAINERS & AMIs, AWS IOT GREENGRASS, ELASTIC INFERENCE, INFERENTIA

Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale



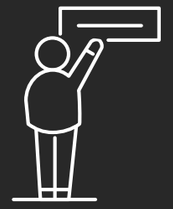
Collect
and prepare
training data



Choose and
optimize your
ML algorithm



Set up and
manage
environments
for training



Train and
tune ML Models



Deploy models
in production

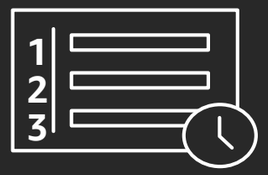


Scale and manage
the production
environment

Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems

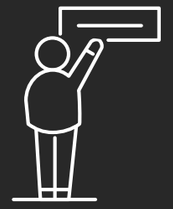
Collect and prepare training data



Choose and optimize your ML algorithm



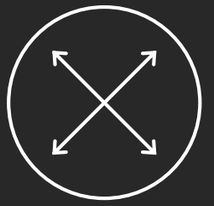
Set up and manage environments for training



Train and tune ML Models



Deploy models in production



Scale and manage the production environment

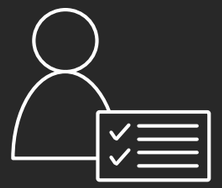
Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems

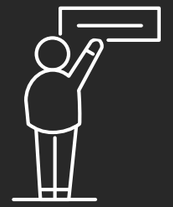
Collect and prepare training data

Built-in, high performance algorithms

Choose and optimize your ML algorithm



Set up and manage environments for training



Train and tune ML Models

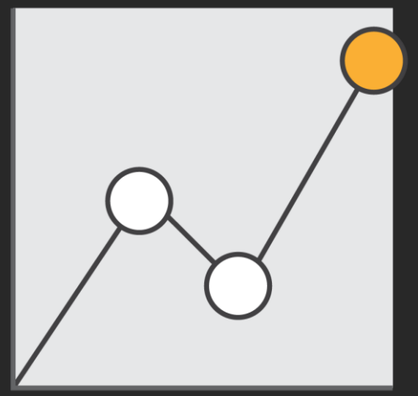


Deploy models in production



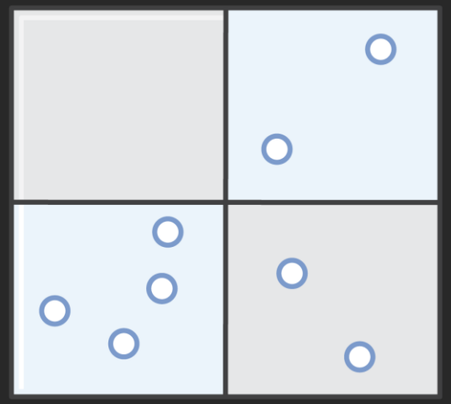
Scale and manage the production environment

Built-in algorithms



XGBoost, FM, Linear Learner,

k-NN for supervised learning



k-Means, PCA, and Random Cut Forest for unsupervised learning

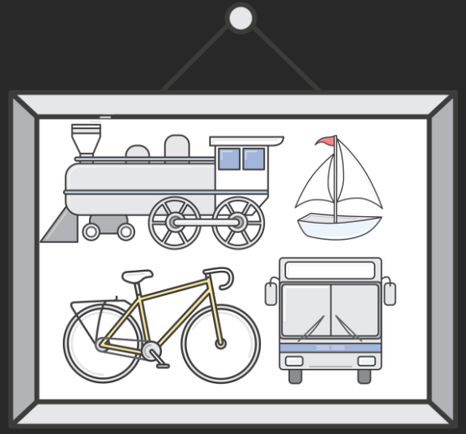


Image classification and object detection for computer vision



LDA, Neural Topic Model, Seq2seq, and Word2Vec for text and NLP

Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems

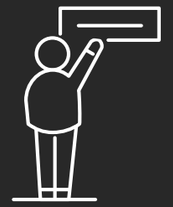
Collect and prepare training data

Built-in, high performance algorithms

Choose and optimize your ML algorithm

One-click training on the highest performing infrastructure

Set up and manage environments for training



Train and tune ML Models



Deploy models in production

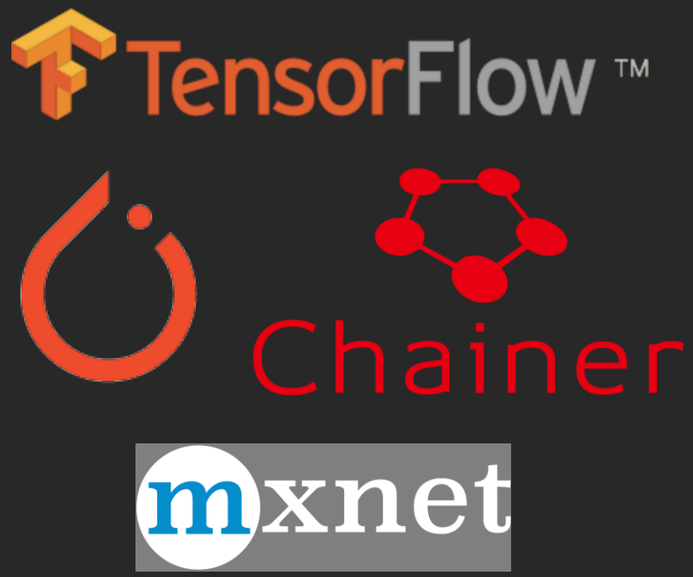


Scale and manage the production environment

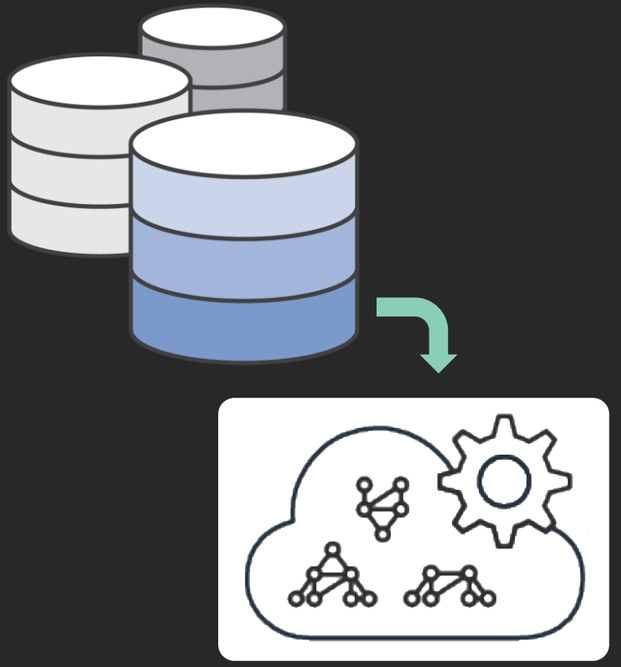
Deep learning framework containers



Sample your data…



… explore and refine models in a single Notebook Instance



Use the same code to train on the full dataset in a cluster of GPU instances…



… deploy to production

Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems

Collect and prepare training data

Built-in, high performance algorithms

Choose and optimize your ML algorithm

One-click training on the highest performing infrastructure

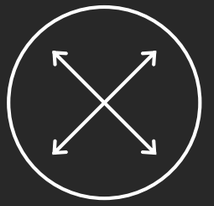
Set up and manage environments for training

Model optimization

Train and tune ML Models

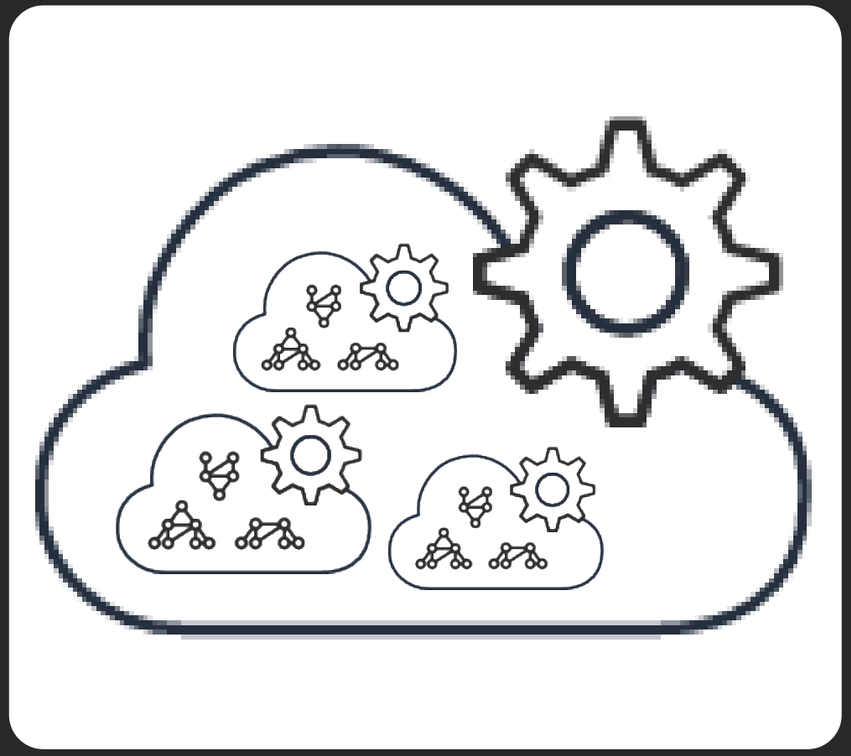


Deploy models in production

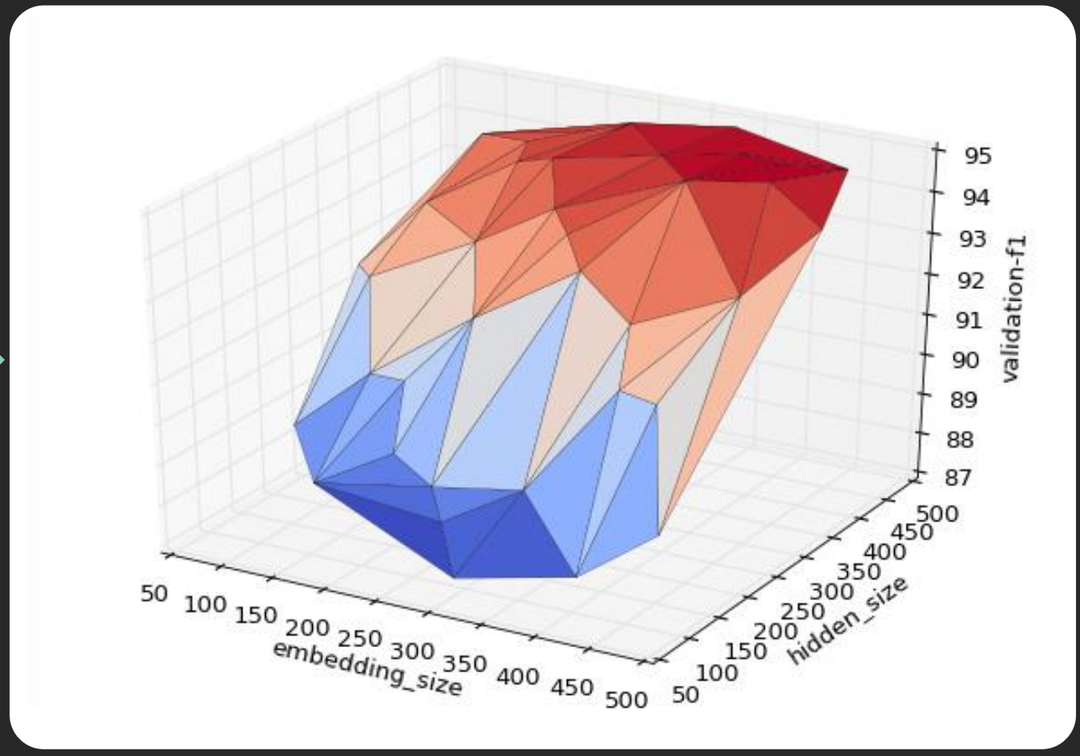


Scale and manage the production environment

Automatic Model Tuning



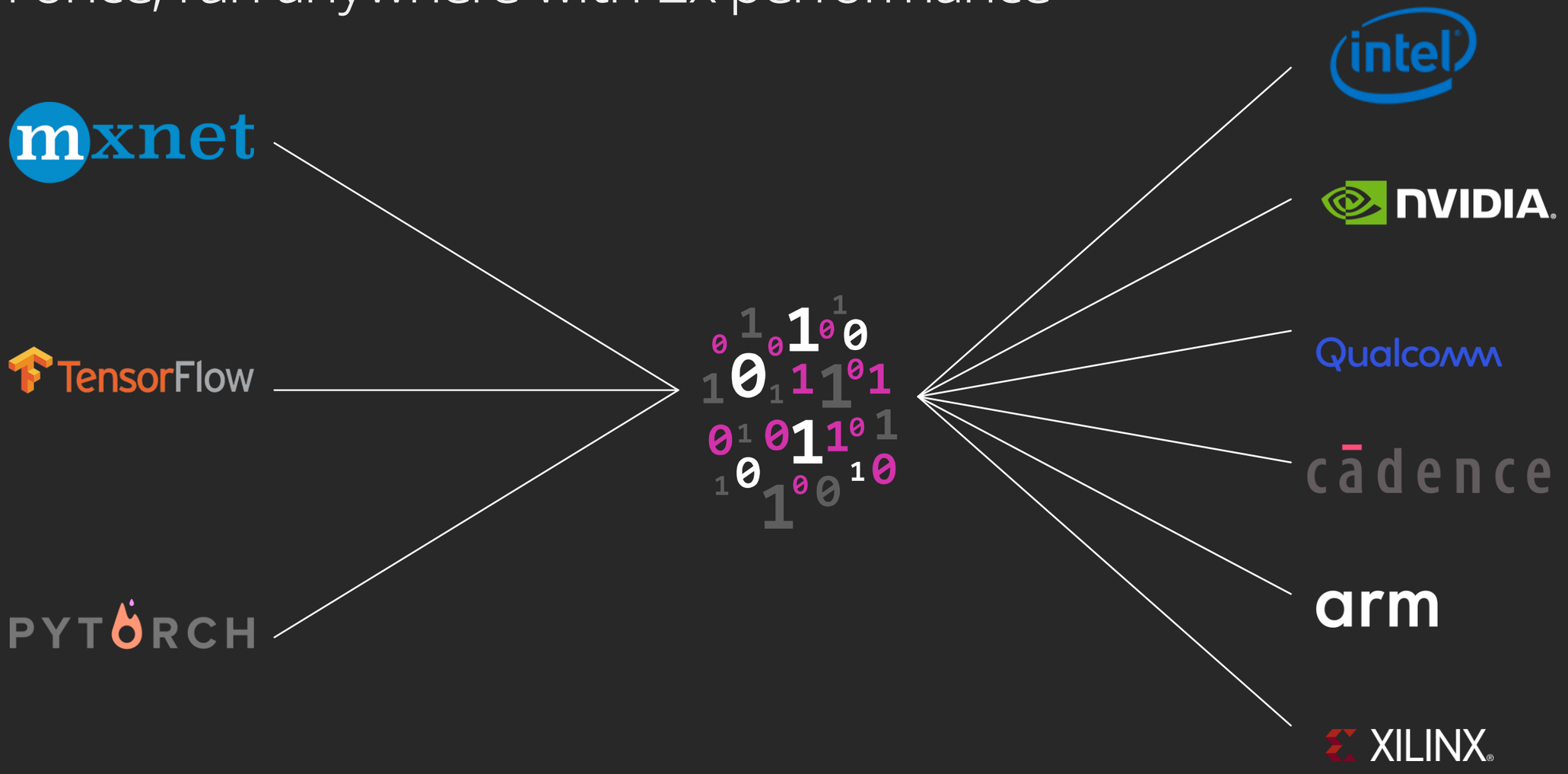
Run a large set of training jobs with varying hyperparameters…



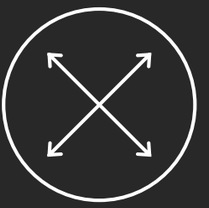
… and search the hyperparameter space for improved accuracy.

Amazon SageMaker Neo

Train once, run anywhere with 2x performance



Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems	Built-in, high performance algorithms	One-click training on the highest performing infrastructure	Model optimization	One-click deployment	
Collect and prepare training data	Choose and optimize your ML algorithm	Set up and manage environments for training	Train and tune ML Models	Deploy models in production	Scale and manage the production environment

Amazon SageMaker:
Build, Train, and Deploy ML Models at Scale

Pre-built notebooks for common problems	Built-in, high performance algorithms	One-click training on the highest performing infrastructure	Model optimization	One-click deployment	Fully managed with auto-scaling for 75% less
Collect and prepare training data	Choose and optimize your ML algorithm	Set up and manage environments for training	Train and tune ML Models	Deploy models in production	Scale and manage the production environment

Overview of BlazingText Algorithm

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

BlazingText Highlights

Provides highly optimized implementations of the Word2vec and text classification algorithms.

BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs

Saurabh Gupta
Amazon Web Services
gsaur@amazon.com

Vineet Khare
Amazon Web Services
vkhare@amazon.com

ABSTRACT

Word2Vec is a popular algorithm used for generating dense vector representations of words in large corpora using unsupervised learning. The resulting vectors have been shown to capture semantic relationships between the corresponding words and are used extensively for many downstream natural language processing (NLP) tasks like sentiment analysis, named entity recognition and machine translation. Most open-source implementations of the algorithm have been parallelized for multi-core CPU architectures including the original C implementation by Mikolov et al. [1] and FastText [2] by Facebook. A few other implementations have attempted to leverage GPU parallelization but at the cost of accuracy and scalability. In this work, we present BlazingText, a highly optimized implementation of word2vec in CUDA, that can leverage multiple GPUs for training. BlazingText can achieve a training speed of up to 43M words/sec on 8 GPUs, which is a 9x speedup over 8-threaded CPU implementations, with minimal effect on the quality of the embeddings.

≈ vec("Man"). This idea has enabled many Natural Language Processing (NLP) algorithms to achieve better performance [3, 4].

The optimization in word2vec is done using Stochastic Gradient Descent (SGD), which solves the problem iteratively; at each step, it picks a pair of words: an input word and a target word either from its window or a random negative sample. It then computes the gradients of the objective function with respect to the two chosen words, and updates the word representations of the two words based on the gradient values. The algorithm then proceeds to the next iteration with a different word pair being chosen.

One of the main issues with SGD is that it is inherently sequential; since there is a dependency between the update from one iteration and the computation in the next iteration (they may happen to touch the same word representations), each iteration must potentially wait for the update from the previous iteration to complete. This does not allow us to use the parallel resources of the hardware.

However, to solve the above issue, word2vec uses Hogwild [5], a scheme where different threads process different word pairs in

BlazingText Highlights

Offers an implementation of the supervised multi-class, multi-label text classification algorithm extending the fastText text classifier to use GPU acceleration with custom CUDA kernels.

BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs

Saurabh Gupta
Amazon Web Services
gsaur@amazon.com

Vineet Khare
Amazon Web Services
vkhare@amazon.com

ABSTRACT

Word2Vec is a popular algorithm used for generating dense vector representations of words in large corpora using unsupervised learning. The resulting vectors have been shown to capture semantic relationships between the corresponding words and are used extensively for many downstream natural language processing (NLP) tasks like sentiment analysis, named entity recognition and machine translation. Most open-source implementations of the algorithm have been parallelized for multi-core CPU architectures including the original C implementation by Mikolov et al. [1] and FastText [2] by Facebook. A few other implementations have attempted to leverage GPU parallelization but at the cost of accuracy and scalability. In this work, we present BlazingText, a highly optimized implementation of word2vec in CUDA, that can leverage multiple GPUs for training. BlazingText can achieve a training speed of up to 43M words/sec on 8 GPUs, which is a 9x speedup over 8-threaded CPU implementations, with minimal effect on the quality of the embeddings.

≈ vec("Man"). This idea has enabled many Natural Language Processing (NLP) algorithms to achieve better performance [3, 4].

The optimization in word2vec is done using Stochastic Gradient Descent (SGD), which solves the problem iteratively; at each step, it picks a pair of words: an input word and a target word either from its window or a random negative sample. It then computes the gradients of the objective function with respect to the two chosen words, and updates the word representations of the two words based on the gradient values. The algorithm then proceeds to the next iteration with a different word pair being chosen.

One of the main issues with SGD is that it is inherently sequential; since there is a dependency between the update from one iteration and the computation in the next iteration (they may happen to touch the same word representations), each iteration must potentially wait for the update from the previous iteration to complete. This does not allow us to use the parallel resources of the hardware.

However, to solve the above issue, word2vec uses Hogwild [5], a scheme where different threads process different word pairs in

BlazingText Highlights

You can train a model on more than a billion words in a couple of minutes using a multi-core CPU or a GPU. And, you achieve performance on par with the state-of-the-art deep learning text classification algorithms.

BlazingText: Scaling and Accelerating Word2Vec using Multiple GPUs

Saurabh Gupta
Amazon Web Services
gsaur@amazon.com

Vineet Khare
Amazon Web Services
vkhare@amazon.com

ABSTRACT

Word2Vec is a popular algorithm used for generating dense vector representations of words in large corpora using unsupervised learning. The resulting vectors have been shown to capture semantic relationships between the corresponding words and are used extensively for many downstream natural language processing (NLP) tasks like sentiment analysis, named entity recognition and machine translation. Most open-source implementations of the algorithm have been parallelized for multi-core CPU architectures including the original C implementation by Mikolov et al. [1] and FastText [2] by Facebook. A few other implementations have attempted to leverage GPU parallelization but at the cost of accuracy and scalability. In this work, we present BlazingText, a highly optimized implementation of word2vec in CUDA, that can leverage multiple GPUs for training. BlazingText can achieve a training speed of up to 43M words/sec on 8 GPUs, which is a 9x speedup over 8-threaded CPU implementations, with minimal effect on the quality of the embeddings.

≈ vec("Man"). This idea has enabled many Natural Language Processing (NLP) algorithms to achieve better performance [3, 4].

The optimization in word2vec is done using Stochastic Gradient Descent (SGD), which solves the problem iteratively; at each step, it picks a pair of words: an input word and a target word either from its window or a random negative sample. It then computes the gradients of the objective function with respect to the two chosen words, and updates the word representations of the two words based on the gradient values. The algorithm then proceeds to the next iteration with a different word pair being chosen.

One of the main issues with SGD is that it is inherently sequential; since there is a dependency between the update from one iteration and the computation in the next iteration (they may happen to touch the same word representations), each iteration must potentially wait for the update from the previous iteration to complete. This does not allow us to use the parallel resources of the hardware.

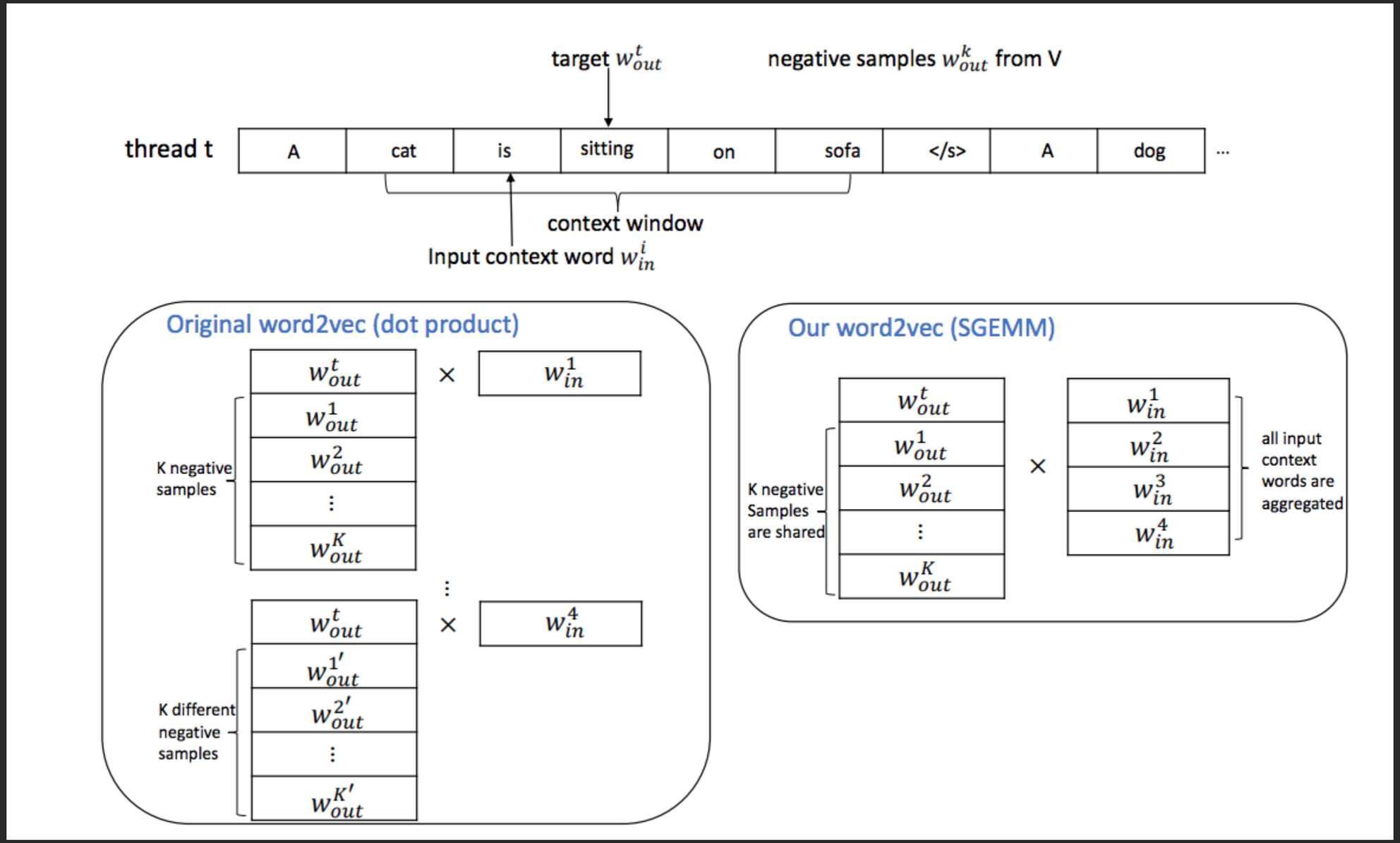
However, to solve the above issue, word2vec uses Hogwild [5], a scheme where different threads process different word pairs in

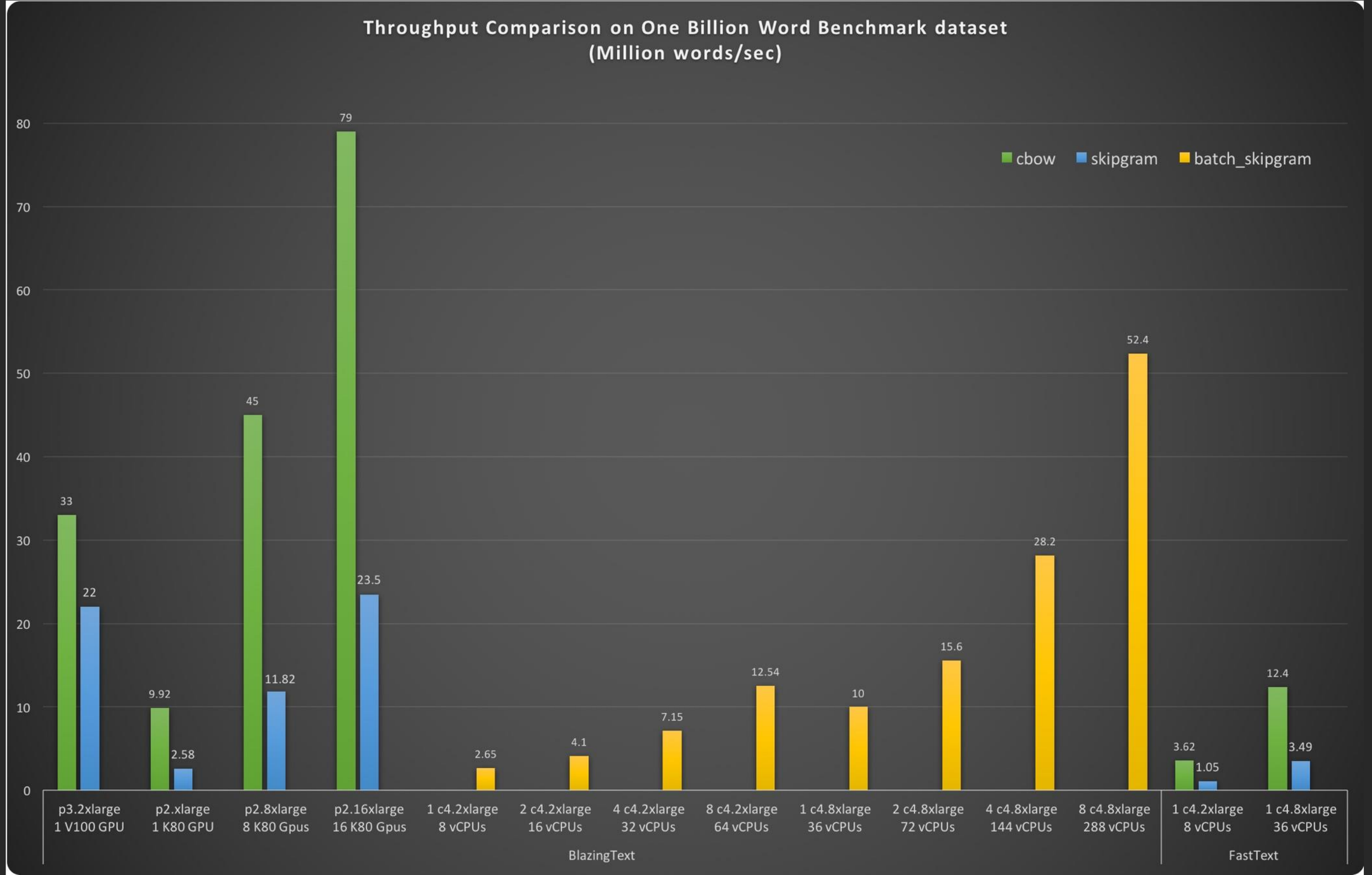
BlazingText Highlights

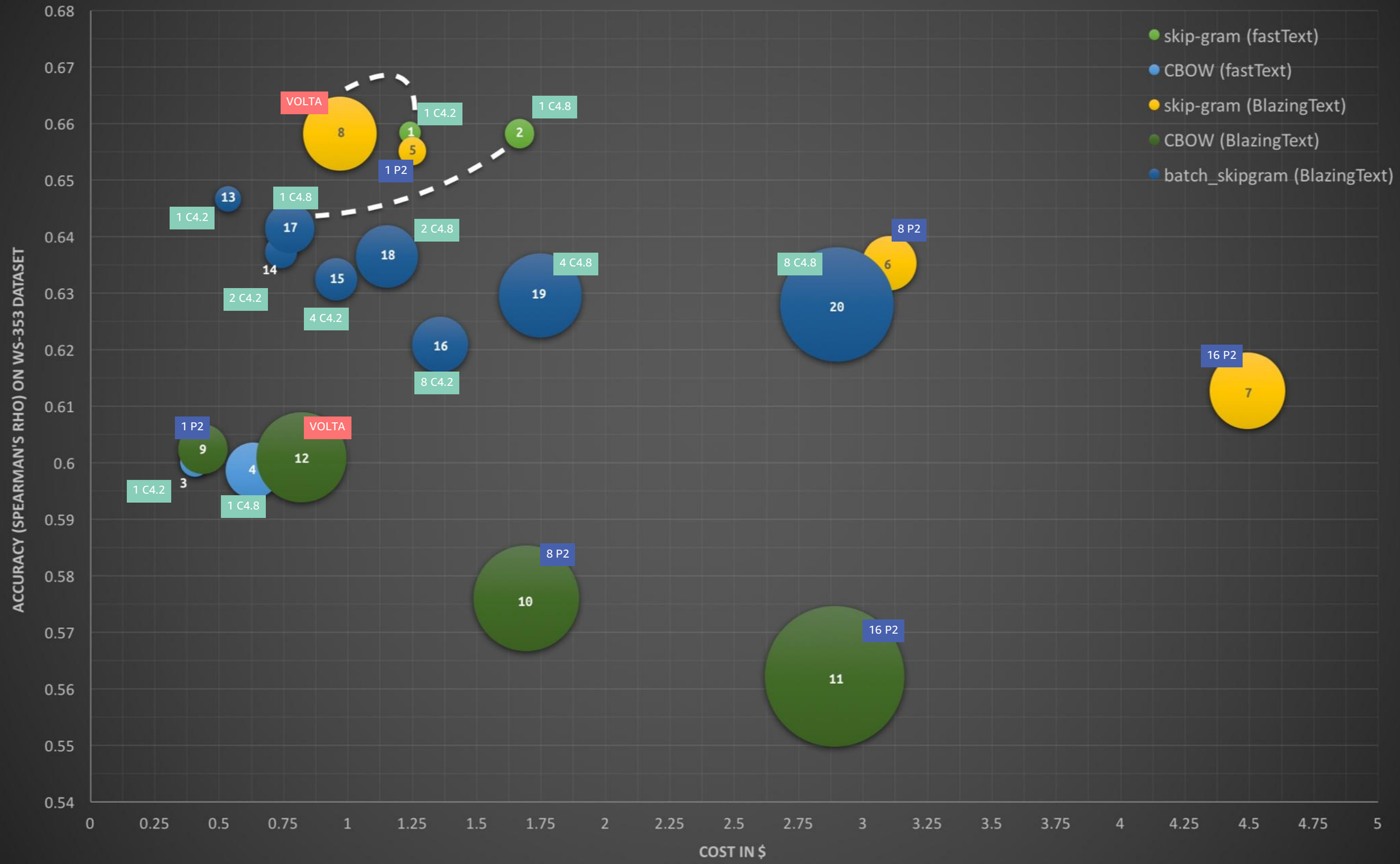
It is the fastest implementation of Word2Vec, available to Amazon SageMaker users on:

- Single CPU instances (like the original C implementation by Mikolov and fastText)
- Single instances with multiple GPUs, P2 or P3. The GPU acceleration is done using highly optimized CUDA kernels.
- Multiple CPU instances (distributed CPU training). Training speed scales almost linearly with number of instances.
- On a single p3.2xlarge (1 Volta V100 GPU) instance, BlazingText can be 21 times faster and 20% cheaper than fastText on a single c4.2xlarge instance.
- For distributed training across multiple CPU nodes, BlazingText can achieve a training speed of up to 50 million words/sec on eight c4.8xlarge instances.
  This is a speedup of 11 times over one c4.8xlarge fastText CPU implementation, which gives a training speed of 4.5 million words/sec.

Optimization on CPU: negative samples sharing




Throughput Comparison on One Billion Word Benchmark dataset
(Million words/sec)
cbow
skipgram
batch_skipgram
80
70
60
50
40
30
20
10
0
33
22
9.92
2.58
45
11.82
79
23.5
2.65
4.1
7.15
12.54
10
15.6
28.2
52.4
3.62
1.05
12.4
3.49
p3.2xlarge
1 V100 GPU
p2.xlarge
1 K80 GPU
p2.8xlarge
8 K80 Gpus
p2.16xlarge
16 K80 Gpus
1 c4.2xlarge
8 vCPUs
2 c4.2xlarge
16 vCPUs
4 c4.2xlarge
32 vCPUs
8 c4.2xlarge
64 vCPUs
1 c4.8xlarge
36 vCPUs
2 c4.8xlarge
72 vCPUs
4 c4.8xlarge
144 vCPUs
8 c4.8xlarge
288 vCPUs
1 c4.2xlarge
8 vCPUs
1 c4.8xlarge
36 vCPUs
BlazingText
FastText

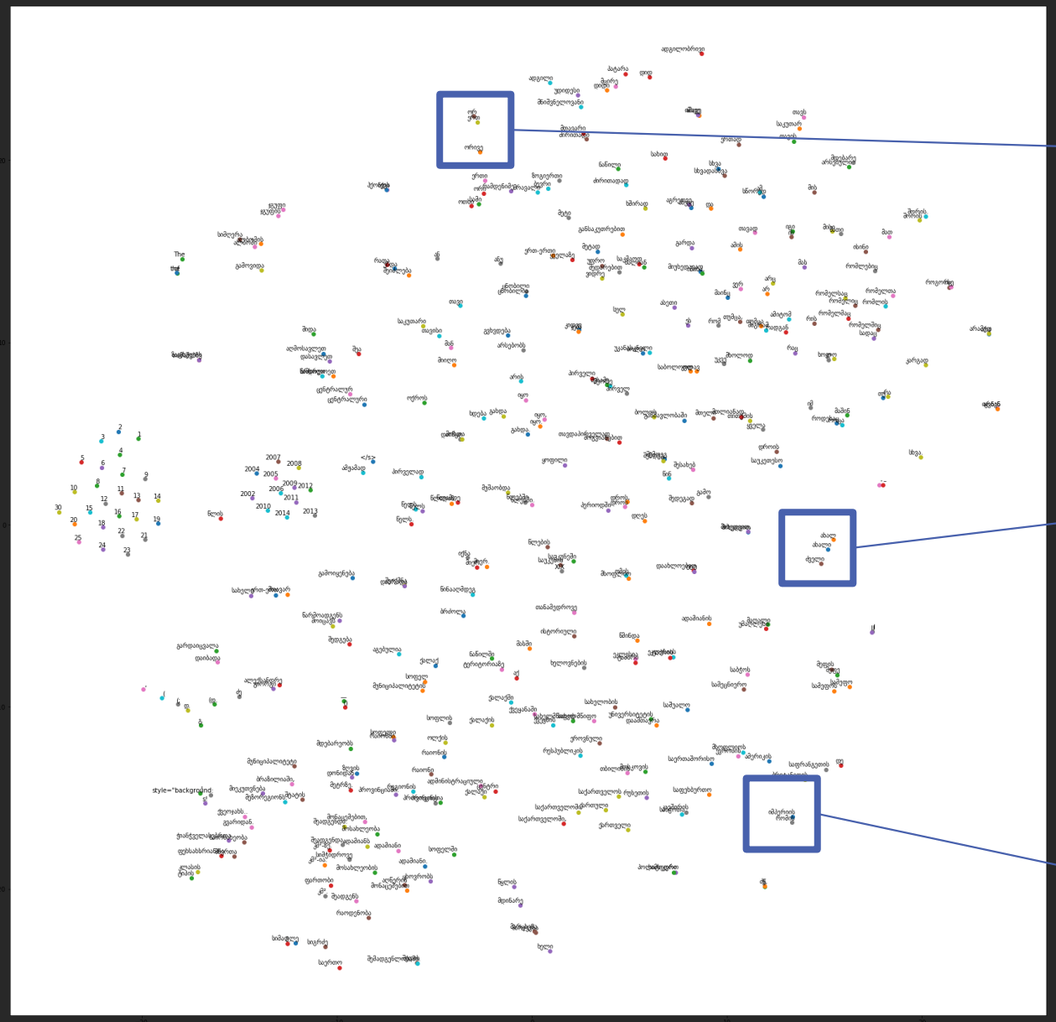

BlazingText Benchmarks (Training dataset: One Billion Word Benchmark)
ACCURACY (SPEARMAN'S RHO) ON WS-353 DATASET
COST IN $
skip-gram (fastText)
CBOW (fastText)
skip-gram (BlazingText)
CBOW (BlazingText)
batch_skipgram (BlazingText)
0.68
0.67
0.66
0.65
0.64
0.63
0.62
0.61
0.6
0.59
0.58
0.57
0.56
0.55
0.54
0
0.25
0.5
0.75
1
1.25
1.5
1.75
2
2.25
2.5
2.75
3
3.25
3.5
3.75
4
4.25
4.5
4.75
5
VOLTA
8
1 C4.2
1
1 C4.8
2
5
1 P2
13
1 C4.8
1 C4.2
17
2 C4.8
14
18
15
4 C4.8
19
2 C4.2
4 C4.2
16
8 C4.2
8 P2
8 C4.8
6
20
16 P2
7
1 P2
9
VOLTA
12
4
3
1 C4.2
1 C4.8
8 P2
10
16 P2
11

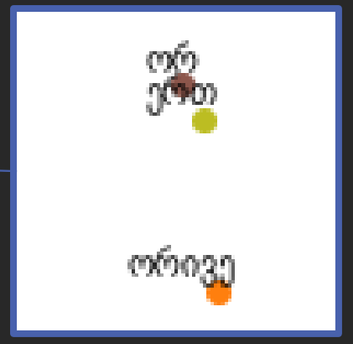
დემო (or Demo)

https://github.com/dbatalov/wikipedia-embedding

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.

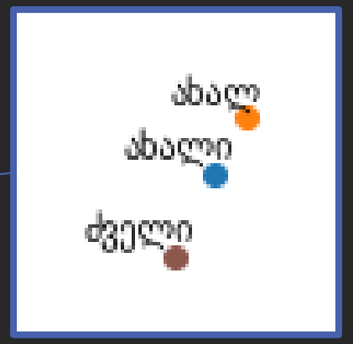
BlazingText: Georgian Word Embedding





ორ
ერთ
ორივე

"two" ("or")
"one" ("ert")
"both" ("orive")



ახალი
ძველი

"new" ("akhali")
"old" ("dzveli")



რომის
იმპერიის

"Roman" ("romis")
"empire" ("imperiis")

Thank you!



Denis V. Batalov, PhD

WW Tech Leader, AI/ML

AWS

© 2019, Amazon Web Services, Inc. or its affiliates. All rights reserved.