Marian NMT

Fast neural machine translation in C++

November 1, 2017

Outline

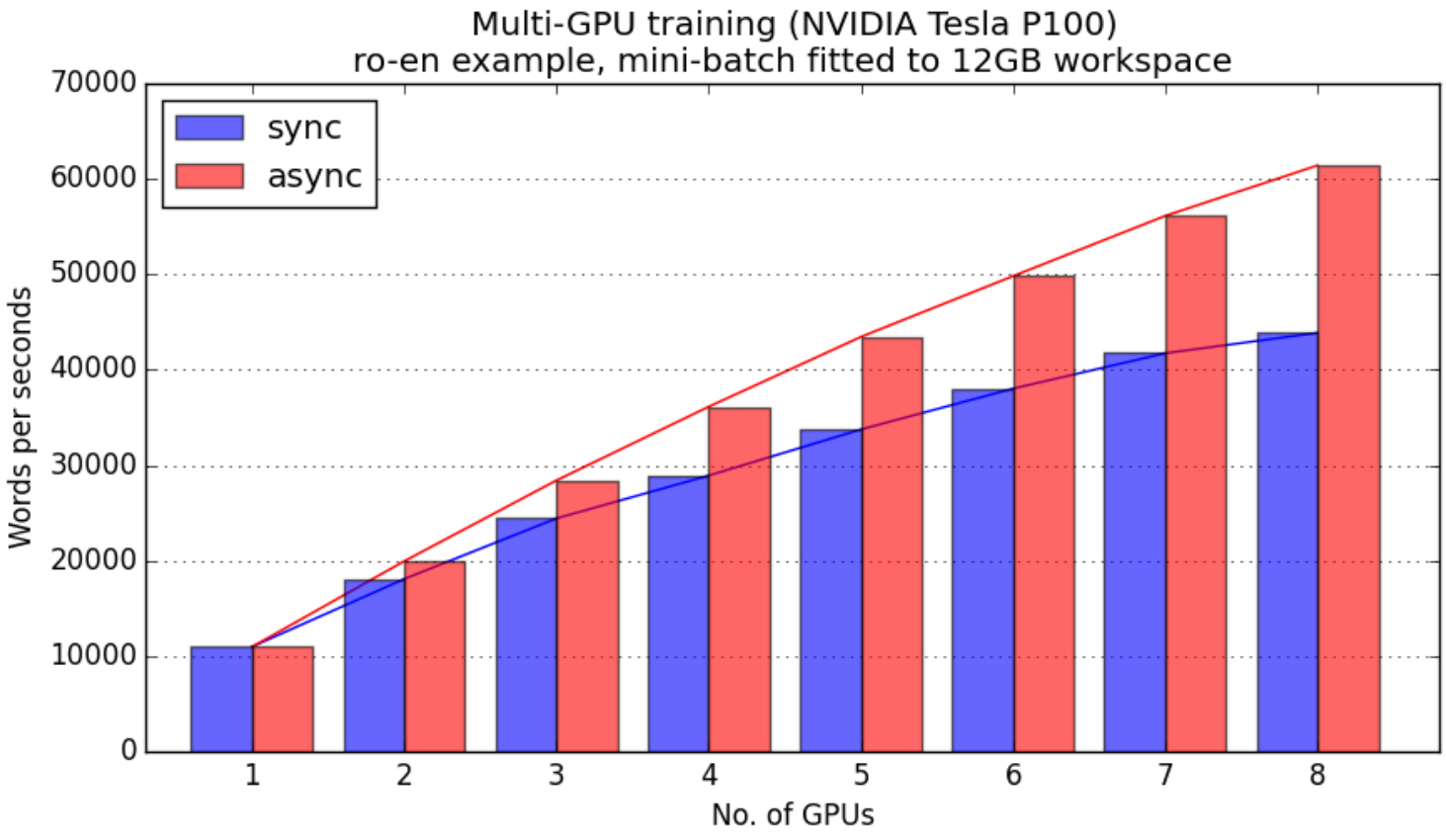
Marian NMT

History:

- Rapid development started after MTM 2016
- A C++ implementation of a reverse-mode automatic differentiation

Overview

- Pure C++ implementation
- Up to 15x faster translation
- Up to 2x faster training
- Multi-GPU training (sync/async) and translation
- Different types of deep and multi-source models


Multi-GPU training (NVIDIA Tesla P100)
ro-en example, mini-batch fitted to 12GB workspace
sync
async
Words per seconds
70000
60000
50000
40000
30000
20000
10000
0
1
2
3
4
5
6
7
8
No. of GPUs

Features

- GPU and LSTM cells
- Deep and multi-source models:
  nematus, *s2s*, *transformer*, *lm*, ...
- Encoder types: *bidirectional*, *uni-bidirectional*, *alternating*
- Layer normalization
- Moving average of parameters
- Decaying learning rate
- Fitting mini-batches
- Residual connections
- Tied embeddings
- Different attention mechanisms
- ...

Marian vs. Amun

amunmt $\rightarrow$ *marian-nmt*:

- AmuNMT: `amun` (CPU and GPU decoding for Nematus models)
- Marian: `marian` (GPU training), `marian-decoder` (translation), `marian-scorer` (rescoring), `marian-server` (web-socket server)

More information

- https://marian-nmt.github.io
- https://github.com/marian-nmt/marian-dev
- https://groups.google.com/forum/#!forum/marian-nmt

Outline

To follow the slides:

- Download the presentation:
  `https://marian-nmt.github.io/materials/marian-nov-17.pdf`
- Browse the code:
  `https://github.com/marian-nmt/marian-dev`
- See MTM 2017 tutorial:
  `https://marian-nmt.github.io/examples/mtm2017/code/`

Expression Graph

https://github.com/marian-nmt/marian-dev/tree/master/src/examples/iris/iris.cpp

Expression graph

- Dynamic graphs (like DyNet, PyTorch)
- A reverse-mode automatic differentiation
- File: `src/graph/expression_graph.h`

```
auto graph = New<ExpressionGraph>();
```

Building an expression graph

```
1   // define the input layer
2   auto x = graph->constant({N, NUM_FEATURES},
3                            init=inits::from_vector(inputData));
4
5   // define the hidden layer
6   auto W1 = graph->param("W1", {NUM_FEATURES, 5}, init=inits::uniform());
7   auto b1 = graph->param("b1", {1, 5}, init=inits::zeros);
8   auto h = tanh(affine(x, W1, b1));
9
10  // define the output layer
11  auto W2 = graph->param("W2", {5, NUM_LABELS}, init=inits::uniform());
12  auto b2 = graph->param("b2", {1, NUM_LABELS}, init=inits::zeros);
13  auto o = affine(h, W2, b2);
```

Building an expression graph

For training

```
// add the cross entropy
auto y = graph->constant({N}, init=inits::from_vector(outputData));
auto cost = mean(cross_entropy(o, y), axis = 0);

// alternatively
// auto cost = -mean(sum(logsoftmax(o) * y, axis=1), axis=0)
```

For decoding

```
// add the softmax layer
auto pred = logsoftmax(o);
```

Nodes

- `Expr constant(Shape shape, Args...);`
- `Expr param(std::string name, Shape shape, Args...);`
- `Expr ones(Args...);`
- `Expr zeros(Args...);`
- `Expr dropout(float prob, Shape shape);`
- ...

Expression operators

- `+`, `-`, `*`, `/`
- `sum`, `mean`, `sqrt`, `log`, `exp`, `logit`
- `relu`, `tanh`, `swish`
- `dot`, `transpose`, `concatenate`, `reshape`, `flatten`
- `rows`, `cols`, `select`
- `affine`, `softmax`, `cross_entropy`
- ...

See file: `src/graph/expression_operators.h`

Function building a graph

```
1  Expr build(Ptr<ExpressionGraph> graph,
2             std::vector<float> inputData,
3             std::vector<float> outputData = {},
4             bool train = false) {
5    graph->clear();
6
7    // define the input layer
8    auto x = graph->constant({N, NUM_FEATURES},
9                             init=inits::from_vector(inputData));
10
11   // define the hidden layer
12   auto W1 = graph->param("W1", {NUM_FEATURES, 5}, init=inits::uniform());
13   auto b1 = graph->param("b1", {1, 5}, init=inits::zeros);
14   auto h = tanh(affine(x, W1, b1));
15
16   // define the output layer
17   auto W2 = graph->param("W2", {5, NUM_LABELS}, init=inits::uniform());
18   auto b2 = graph->param("b2", {1, NUM_LABELS}, init=inits::zeros);
19   auto o = affine(h, W2, b2);
20
21   if(train) {
22     auto y = graph->constant({N}, init=inits::from_vector(outputData));
23     return mean(cross_entropy(o, y), axis = 0);
24   } else {
25     return logsoftmax(o);
26   }
27 }
```

Training loop

```
1  {
2    auto graph = New<ExpressionGraph>();
3    graph->setDevice(0);
4    graph->reserveWorkspaceMB(128);
5
6    // choose optimizer and initial learning rate
7    auto opt = Optimizer<Adam>(0.005);
8
9    for(size_t epoch = 1; epoch <= MAX_EPOCHS; ++epoch) {
10     shuffleData(trainX, trainY);
11
12     // build the classifier
13     auto cost = build(graph, trainX, trainY, true);
14
15     // train the network and update weights
16     graph->forward();
17     graph->backward();
18     opt->update(graph);
19
20     std::cout << "Epoch: " << epoch << " Cost: " << cost->scalar()
21               << std::endl;
22   }
23 }
```

Prediction

```
1  auto probs = build(graph, testX);
2  // debug(probs, "Probs:")
3
4  // run classifier
5  graph->forward();
6
7  // extract predictions
8  std::vector<float> preds(testY.size());
9  probs->val()->get(preds);
10
11 std::cout << "Accuracy: " << calculateAccuracy(preds, testY)
12           << std::endl;
```

Complete examples:

- Iris: `src/examples/iris`
- MNIST: `src/examples/mnist`

Expression operators

https://github.com/marian-nmt/marian-dev/tree/master/src/graph/node_operators_unary.h

New activation function

Swish (Ramachandran et al. 2017):

$$f(x) = x \cdot \sigma(x)$$

$$f'(x) = f(x) + \sigma(x)(1 - f(x))$$

Skeleton

```
// src/graph/expression_operators.h
Expr swish(Expr a);

// src/graph/expression_operators.cu
Expr swish(Expr a) {
  return Expression<SwishNodeOp>(a);
}
```

```
// src/graph/node_operators_unary.h
struct SwishNodeOp : public UnaryNodeOp {
  template <typename... Args>
  SwishNodeOp(Args... args) : UnaryNodeOp(args...) {}

  NodeOps forwardOps() { return {/*...*/}; }
  NodeOps backwardOps() { return {/*...*/}; }

  const std::string type() { return "swish"; }
};
```

Forward step

Swish:

$$f(x) = x \cdot \sigma(x)$$

```
1  NodeOps forwardOps() {
2    return {NodeOp(Element(_1 = _2 * Sigma(_2),
3                           val_,
4                           child(0)->val()
5                         ))};
6  }
```

Backward step

Derivative:

$$f'(x) = f(x) + \sigma(x)(1 - f(x))$$

We need:

$$\frac{\partial J}{\partial x} \mathrel{+}= \frac{\partial J}{\partial f} \cdot f'(x)$$

```
1  NodeOps backwardOps() {
2    return {NodeOp(Add(_1 * (_3 + Sigma(_2) * (1.f - _3)),
3                       child(0)->grad(),  // dJ/dx
4                       adj_,              // _1 := dJ/df
5                       child(0)->val(),   // _2 := x
6                       val_               // _3 := f(x) = x*sigma(x)
7                   ))};
8  }
```

Alternative way: write a kernel/thrust function

See ReLU or PReLU operators

Sutskever-style model

`https://github.com/marian-nmt/marian-dev/tree/tutorial-nov-17/src/models/sutskever.h`

MTM2017 tutorial

https://marian-nmt.github.io/examples/mtm2017/code

For lazy people:

```
cd marian-dev
git fetch origin tutorial-nov-17
git checkout tutorial-nov-17
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release
make -j8
```

Encoder

```
1  // skeleton code for encoder
2  class EncoderSutskever : public EncoderBase {
3  public:
4    EncoderSutskever(Ptr<Options> options) : EncoderBase(options) {}
5
6    Ptr<EncoderState> build(Ptr<ExpressionGraph> graph,
7                            Ptr<data::CorpusBatch> batch) {
8      return New<EncoderState>(nullptr, nullptr, batch);
9    }
10
11   void clear() {}
12 };
```

Decoder

```
1  // skeleton code for decoder
2  class DecoderSutskever : public DecoderBase {
3  public:
4    DecoderSutskever(Ptr<Options> options) : DecoderBase(options) {}
5
6    virtual Ptr<DecoderState> startState(
7        Ptr<ExpressionGraph> graph,
8        Ptr<data::CorpusBatch> batch,
9        std::vector<Ptr<EncoderState>>& encStates) {
10
11     rnn::States startStates;
12     return New<DecoderState>(startStates, nullptr, encStates);
13   }
14
15   virtual Ptr<DecoderState> step(Ptr<ExpressionGraph> graph,
16                                  Ptr<DecoderState> state) {
17     rnn::States decoderStates;
18     return New<DecoderState>(decoderStates,
19                              nullptr,
20                              state->getEncoderStates());
21   }
22
23   void clear() {}
24 };
```

Register the encoder and the decoder

```
#include "models/sutskever.h"

// ...

Ptr<EncoderBase> EncoderFactory::construct() {
  if(options_->get<std::string>("type") == "sutskever")
    return New<EncoderSutskever>(options_);
  // ...

Ptr<DecoderBase> DecoderFactory::construct() {
  if(options_->get<std::string>("type") == "sutskever")
    return New<DecoderSutskever>(options_);
  // ...
```

Construct the encoder-decoder model

```
1  Ptr<ModelBase> by_type(std::string type, Ptr<Options> options) {
2    if(type == "sutskever") {
3      return models::encoder_decoder()(options)
4          .push_back(models::encoder()("type", "sutskever"))
5          .push_back(models::decoder()("type", "sutskever"))
6          .construct();
7    }
8    // ...
```

Encoder::build

Source embeddings

```
1  // create source embeddings
2  int dimVoc = opt<std::vector<int>>("dim-vocabs")[batchIndex_];
3  auto embeddings = embedding(graph)
4                    ("prefix", prefix_ + "_Wemb")
5                    ("dimVocab", dimVoc)
6                    ("dimEmb", opt<int>("dim-emb"))
7                        .construct();
```

More on `embeddings(Ptr<ExpressionGraph>)`

Encoder::build

Embedding look-up

```
1  // select embeddings that occur in the batch
2  Expr batchEmbeddings, batchMask;
3  std::tie(batchEmbeddings, batchMask)
4      = EncoderBase::lookup(embeddings, batch, encoderIndex);
```

Encoder::build

Backward encoder RNN

```
1  // backward RNN for encoding
2  float dropoutRnn = inference_ ? 0 : opt<float>("dropout-rnn");
3  auto rnnBw = rnn::rnn(graph)
4                ("type", "lstm")
5                ("prefix", prefix_)
6                ("direction", rnn::dir::backward)
7                ("dimInput", opt<int>("dim-emb"))
8                ("dimState", opt<int>("dim-rnn"))
9                ("dropout", dropoutRnn)
10               ("layer-normalization", opt<bool>("layer-normalization"))
11                   .push_back(rnn::cell(graph))
12                   .construct();
13
14 auto context = rnnBw->transduce(batchEmbeddings, batchMask);
```

Decoder::startState

Setting the start state for decoding

```
virtual Ptr<DecoderState> startState(
    Ptr<ExpressionGraph> graph,
    Ptr<data::CorpusBatch> batch,
    std::vector<Ptr<EncoderState>>& encStates) {
  using namespace keywords;

  // use first encoded word as start state
  auto start = marian::step(encStates[0]->getContext(), 0, 2);

  rnn::States startStates({{start, start}});
  return New<DecoderState>(startStates, nullptr, encStates);
}
```

Decoder::step

Shifted embeddings

```
1  auto embeddings = state->getTargetEmbeddings();
```

Decoder::step

```
1  // forward RNN for decoder
2  float dropoutRnn = inference_ ? 0 : opt<float>("dropout-rnn");
3  auto rnn = rnn::rnn(graph)
4              ("type", "lstm")
5              ("prefix", prefix_)
6              ("dimInput", opt<int>("dim-emb"))
7              ("dimState", opt<int>("dim-rnn"))
8              ("dropout", dropoutRnn)
9              ("layer-normalization", opt<bool>("layer-normalization"))
10             .push_back(rnn::cell(graph))
11             .construct();
12
13 // apply RNN to embeddings, initialized with encoder context
14 // mapped into decoder space
15 auto decoderContext = rnn->transduce(embeddings, state->getStates());
16
17 // retrieve the last state per layer. They are required during
18 // translation in order to continue decoding for the next word
19 rnn::States decoderStates = rnn->lastCellStates();
```

Decoder::step

Deep output (2-layers)

```
1  // construct deep output multi-layer network layer-wise
2  auto layer1 = mlp::dense(graph)
3                ("prefix", prefix_ + "_ff_logit_l1")
4                ("dim", opt<int>("dim-emb"))
5                ("activation", mlp::act::tanh);
6  int dimTrgVoc = opt<std::vector<int>>("dim-vocabs").back();
7  auto layer2 = mlp::dense(graph)
8                ("prefix", prefix_ + "_ff_logit_l2")
9                ("dim", dimTrgVoc);
10
11 // assemble layers into MLP and apply to embeddings, decoder context
12 // and aligned source context
13 auto logits = mlp::mlp(graph)
14               .push_back(layer1)
15               .push_back(layer2)
16               ->apply(embeddings, decoderContext);
```

Decoder::step

Return the decoder state

```
1  // return unnormalized(!) probabilities
2  return New<DecoderState>(decoderStates,
3                           logits,
4                           state->getEncoderStates());
```

Recompile:

```
cd build
make -j
cd ..
```

Train:

```
./build/marian \
  --type sutskever \
  -t data/corpus.bpe.ro data/corpus.bpe.en \
  -v data/vocab.ro.yml data/vocab.en.yml \
  -m model/model.npz
```

Outline

Transformer

Transformer (Attention is All You Need, Vaswani et al. 2017)

New in Marian:

- Transformer model
- New dropouts
- Google-style learning rate warm-up
- Label smoothing

Experiments

Settings:

- WMT2017 en-de training data
- Problem-set from the `tensor2tensor` repo by Google: 36,000 common BPE subwords for both languages

Three modifications:

- Highway connections instead of skip-connections
- Averaging parameters
- Swish function

Results

System	2013	2014	2015	2016
Edinburgh Deep RNN (Micelli Barone et al. 2017)	—	23.4	26.0	31.0
Transformer 12-layers (Ramachandran et al. 2017)	26.1*	27.8*	29.8*	33.3*

* Measured with multi-bleu.perl, rest with mteval-v13a.pl

Results

System	2013	2014	2015	2016
Edinburgh Deep RNN (Micelli Barone et al. 2017)	—	23.4	26.0	31.0
Transformer 12-layers (Ramachandran et al. 2017)	26.1*	27.8*	29.8*	33.3*
Marian Transformer 6-layers (195,000 it.)	25.4*	26.5	29.5	33.2

* Measured with multi-bleu.perl, rest with mteval-v13a.pl

Results

System	2013	2014	2015	2016
Edinburgh Deep RNN (Micelli Barone et al. 2017)	—	23.4	26.0	31.0
Transformer 12-layers (Ramachandran et al. 2017)	26.1*	27.8*	29.8*	33.3*
Marian Transformer 6-layers (195,000 it.)	25.4*	26.5	29.5	33.2
Marian Edinburgh-style Deep RNN (95,000 it.)	24.6*	24.5	28.1	32.4

* Measured with multi-bleu.perl, rest with mteval-v13a.pl

Outline

Future work

- Numpy-style shape (finished)
- Batched decoding
- Facebook conv2conv model
- Moses scorers
- Reinforcement learning / Minimum error rate training
- CPU version
- ...

Any suggestions?

M. Junczys-Dowmunt, T. Dwojak, and H. Hoang. Is neural machine translation ready for deployment? a case study on 30 translation directions. In *Proceedings of the 9th International Workshop on Spoken Language Translation (IWSLT)*, 2016. URL `http://workshop2016.iwslt.org/downloads/IWSLT_2016_paper_4.pdf`.

A. V. Miceli Barone, J. Helcl, R. Sennrich, B. Haddow, and A. Birch. Deep architectures for neural machine translation. Association for Computational Linguistics, 2017. URL `http://aclweb.org/anthology/W17-4710`.

P. Ramachandran, B. Zoph, and Q. V. Le. Searching for activation functions. *arXiv.org*, 2017. URL `https://arxiv.org/pdf/1710.05941.pdf`.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017. URL `https://arxiv.org/pdf/1706.03762.pdf`.