Marian NMT

Fast neural machine translation in C++

November 1, 2017

Outline

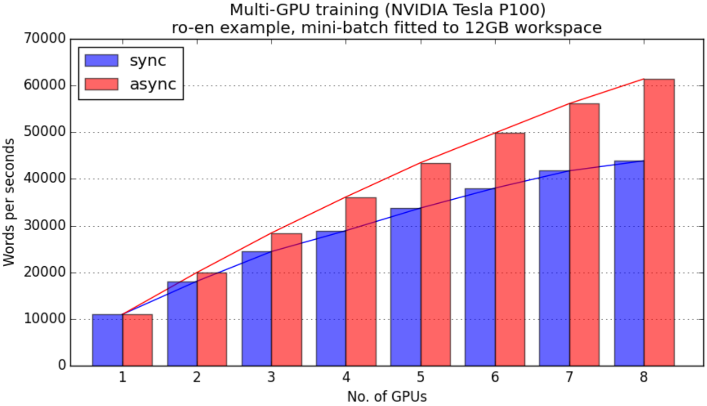
Marian NMT

History:

- Rapid development started after MTM 2016
- C++ implementation of 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
No. of GPUs
0
10000
20000
30000
40000
50000
60000
70000
1
2
3
4
5
6
7
8

Features

- GRU 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)
- Reverse-mode automatic differentiation
- File: `src/graph/expression_graph.h`

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

Building an expression graph

```
// define the input layer
auto x = graph->constant({N, NUM_FEATURES},
                         init=inits::from_vector(inputData));

// define a hidden layer
auto W1 = graph->param("W1", {NUM_FEATURES, 5}, init=inits::uniform());
auto b1 = graph->param("b1", {1, 5}, init=inits::zeros);
auto h = tanh(affine(x, W1, b1));

// define the output layer
auto W2 = graph->param("W2", {5, NUM_LABELS}, init=inits::uniform());
auto b2 = graph->param("b2", {1, NUM_LABELS}, init=inits::zeros);
auto o = affine(h, W2, b2);
```

Building an expression graph

For training

```
// add the cross entropy layer
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`

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`

Unification with Numpy in progress, e.g. `atleast_2d(x)`

Building a graph

```
Expr build(Ptr<ExpressionGraph> graph,
           std::vector<float> inputData,
           std::vector<float> outputData = {},
           bool train = false) {
  graph->clear();

  // define the input layer
  auto x = graph->constant({N, NUM_FEATURES},
                           init=inits::from_vector(inputData));

  // define a hidden layer
  auto W1 = graph->param("W1", {NUM_FEATURES, 5}, init=inits::uniform());
  auto b1 = graph->param("b1", {1, 5}, init=inits::zeros);
  auto h = tanh(affine(x, W1, b1));

  // define the output layer
  auto W2 = graph->param("W2", {5, NUM_LABELS}, init=inits::uniform());
  auto b2 = graph->param("b2", {1, NUM_LABELS}, init=inits::zeros);
  auto o = affine(h, W2, b2);

  if(train) {
    auto y = graph->constant({N}, init=inits::from_vector(outputData));
    return mean(cross_entropy(o, y), axis = 0);
  } else {
    return logsoftmax(o);
  }
}
```

Training loop

```
{
  auto graph = New<ExpressionGraph>();
  graph->setDevice(0);
  graph->reserveWorkspaceMB(128);

  // choose optimizer and initial learning rate
  auto opt = Optimizer<Adam>(0.005);

  for(size_t epoch = 1; epoch <= MAX_EPOCHS; ++epoch) {
    shuffleData(trainX, trainY);

    // build the classifier
    auto cost = build(graph, trainX, trainY, true);

    // train the network and update weights
    graph->forward();
    graph->backward();
    opt->update(graph);

    std::cout << "Epoch: " << epoch << " Cost: " << cost->scalar()
              << std::endl;
  }
}
```

Prediction

```cpp
auto probs = build(graph, testX);
// debug(probs, "Probs:")

// run classifier
graph->forward();

// extract predictions
std::vector<float> preds(testY.size());
probs->val()->get(preds);

std::cout << "Accuracy: " << calculateAccuracy(preds, testY)
          << 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:

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

```
NodeOps forwardOps() {
  return {NodeOp(Element(_1 = _2 * Sigma(_2),
                         val_,
                         child(0)->val()
                     ))};
}
```

Backward step

Derivative:

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

We need:

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

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

Alternative way: write new 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

```cpp
// skeleton code for encoder
class EncoderSutskever : public EncoderBase {
public:
  EncoderSutskever(Ptr<Options> options) : EncoderBase(options) {}

  Ptr<EncoderState> build(Ptr<ExpressionGraph> graph,
                          Ptr<data::CorpusBatch> batch) {
    return New<EncoderState>(nullptr, nullptr, batch);
  }

  void clear() {}
};
```

Decoder

```
// skeleton code for decoder
class DecoderSutskever : public DecoderBase {
public:
  DecoderSutskever(Ptr<Options> options) : DecoderBase(options) {}

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

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

  virtual Ptr<DecoderState> step(Ptr<ExpressionGraph> graph,
                                 Ptr<DecoderState> state) {
    rnn::States decoderStates;
    return New<DecoderState>(decoderStates,
                             nullptr,
                             state->getEncoderStates());
  }

  void clear() {}
};
```

Register the encoder and the decoder

```cpp
#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

```
Ptr<ModelBase> by_type(std::string type, Ptr<Options> options) {
  if(type == "sutskever") {
    return models::encoder_decoder()(options)
        .push_back(models::encoder()("type", "sutskever"))
        .push_back(models::decoder()("type", "sutskever"))
        .construct();
  }
  // ...
```

Encoder::build

Source embeddings

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

More on `embeddings(Ptr<ExpressionGraph>)`

Encoder::build

Embedding look-up

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

Encoder::build

Backward encoder RNN

```cpp
// backward RNN for encoding
float dropoutRnn = inference_ ? 0 : opt<float>("dropout-rnn");
auto rnnBw = rnn::rnn(graph)
               ("type", "lstm")
               ("prefix", prefix_)
               ("direction", rnn::dir::backward)
               ("dimInput", opt<int>("dim-emb"))
               ("dimState", opt<int>("dim-rnn"))
               ("dropout", dropoutRnn)
               ("layer-normalization", opt<bool>("layer-normalization"))
                   .push_back(rnn::cell(graph))
                   .construct();

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

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

Decoder::step

```
// forward RNN for decoder
float dropoutRnn = inference_ ? 0 : opt<float>("dropout-rnn");
auto rnn = rnn::rnn(graph)
             ("type", "lstm")
             ("prefix", prefix_)
             ("dimInput", opt<int>("dim-emb"))
             ("dimState", opt<int>("dim-rnn"))
             ("dropout", dropoutRnn)
             ("layer-normalization", opt<bool>("layer-normalization"))
             .push_back(rnn::cell(graph))
             .construct();

// apply RNN to embeddings, initialized with encoder context
// mapped into decoder space
auto decoderContext = rnn->transduce(embeddings, state->getStates());

// retrieve the last state per layer. They are required during
// translation in order to continue decoding for the next word
rnn::States decoderStates = rnn->lastCellStates();
```

Decoder::step

Deep output (2-layers)

```cpp
// construct deep output multi-layer network layer-wise
auto layer1 = mlp::dense(graph)
              ("prefix", prefix_ + "_ff_logit_l1")
              ("dim", opt<int>("dim-emb"))
              ("activation", mlp::act::tanh);
int dimTrgVoc = opt<std::vector<int>>("dim-vocabs").back();
auto layer2 = mlp::dense(graph)
              ("prefix", prefix_ + "_ff_logit_l2")
              ("dim", dimTrgVoc);

// assemble layers into MLP and apply to embeddings, decoder context
// and aligned source context
auto logits = mlp::mlp(graph)
              .push_back(layer1)
              .push_back(layer2)
              ->apply(embeddings, decoderContext);
```

Decoder::step

Return the decoder state

```
// return unnormalized(!) probabilities
return New<DecoderState>(decoderStates,
                          logits,
                          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
- Google-style learning rate warm-upG
- 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

Newest results: `https://github.com/marian-nmt/marian-dev/issues/116`

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

Newest results: `https://github.com/marian-nmt/marian-dev/issues/116`

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

Newest results: `https://github.com/marian-nmt/marian-dev/issues/116`

Outline

Future work

- Numpy-style shape (finished) and functions
- Documentation (you are all welcome to help)
- Cleaning low-level GPU code
- 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.