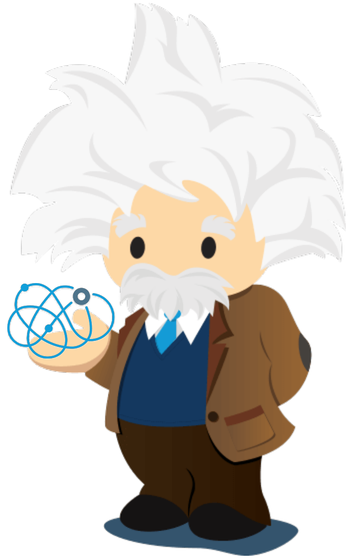
Matchbox

automatic batching for imperative deep learning

James Bradbury

NVIDIA GTC, 2018/3/28



Roadmap

- Imperative deep learning
- How manual batching works
- Other tools for automatic batching
- The Matchbox approach: dispatch and control flow transformation
- Examples

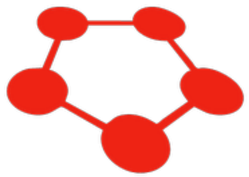
Salesforce Einstein

Imperative Deep Learning

- The last few years has seen the rise of frameworks that allow researchers to **write their models directly as code**

- This is more familiar and ergonomic, and allows programmers to use all the facilities of the language they're programming in (e.g., control flow and debuggers)

autograd



∂y/net

GLUON

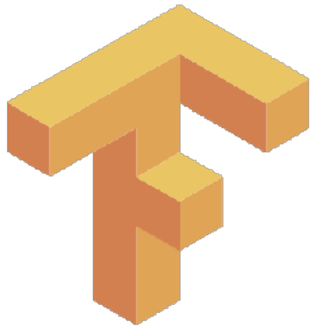
TF Eager

Chainer

PYTORCH

Flux

Imperative Deep Learning



```
cond = lambda i, h: i < tf.shape(words)[0]
cell = lambda i, h: rnn_unit(words[i], h)
i = 0
_, h = tf.while_loop(cond, cell, (i, h0))
```

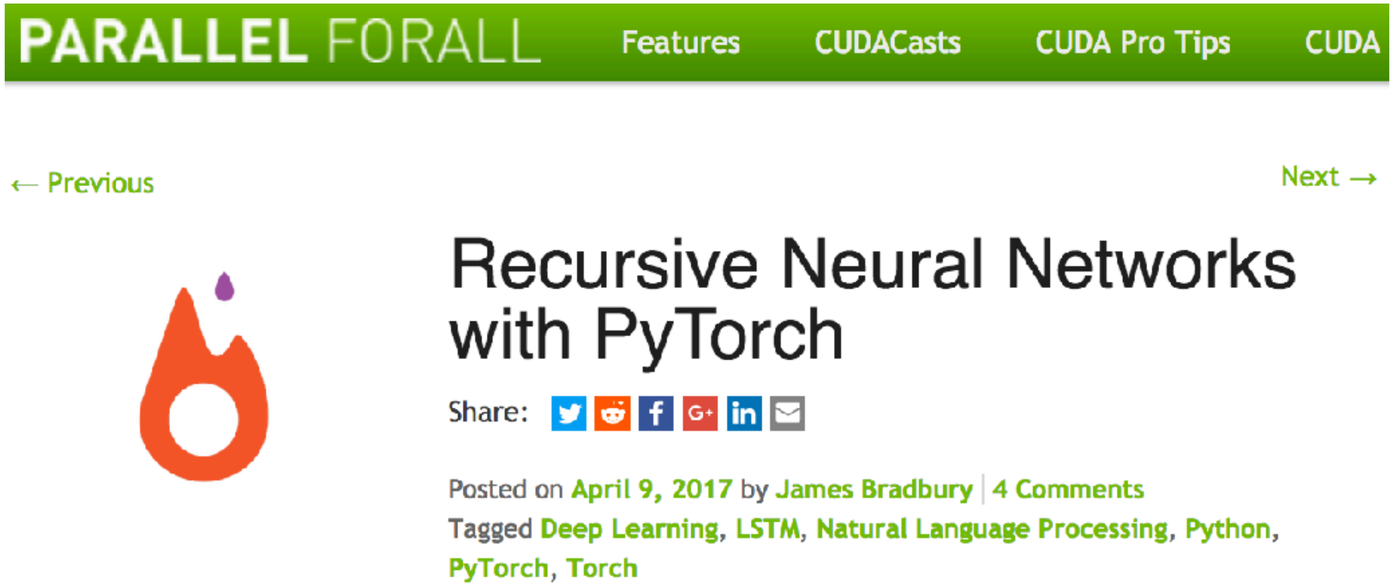
```
h = h0
for word in words:
    h = rnn_unit(word, h)
```

Imperative Deep Learning

...is based on an overstatement.

Salesforce Einstein

An example of this overstatement



"Recursive neural networks are a good demonstration of PyTorch's flexibility"

Salesforce Einstein

Imperative Deep Learning

...is based on an overstatement.

The problem is that **this code doesn't actually work**:

```
h = h0
for word in words:
    h = rnn_unit(word, h)
```

Salesforce Einstein

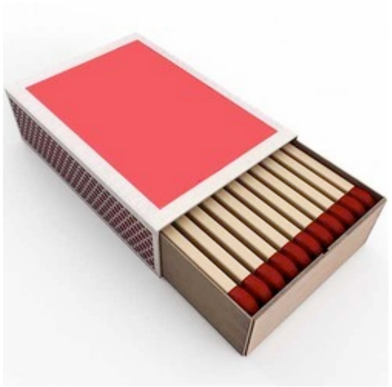
Batching in Deep Learning

Why? Because it's written for a single example (a sequence of words) but **deep learning models usually run on batches of examples**.

This is essential for e.g. taking full advantage of GPU parallelism.

Batching in Deep Learning

Code like the simple `for` loop would be more likely to work if batches looked like this:



But often they look like this, even if programmers intentionally batch together examples with similar properties (here, length):

Batching in Deep Learning

So users of imperative deep learning frameworks must manually modify their code to operate on batches rather than single examples.

This involves “padding” examples so that every batch is a full tensor and “masking” away padding values so they don’t affect computations.

This is hard to get right and even harder to debug, since mistakes lead to silently wrong behavior rather than compile- or run-time errors.

https://github.com/salesforce/nonauto-nmt/blob/master/model.py

mask 1/232

```
        masks = self.prepare_masks(targets) if masks is None else masks
        return targets, masks

    def prepare_masks(self, inputs):
        if inputs.ndimension() == 2:
            masks = (inputs.data != self.field.vocab.stoi['<pad>']).float()
        else:
            masks = (inputs.data[:, :, self.field.vocab.stoi['<pad>']] != 1).float()
        return masks

    def encoding(self, encoder_inputs, encoder_masks):
        return self.encoder(encoder_inputs, encoder_masks)

    def quick_prepare(self, batch, distillation=False, inputs=None, targets=None,
                      input_masks=None, target_masks=None, source_masks=None):
        inputs,  input_masks   = self.prepare_inputs(batch, inputs, distillation, input_masks)     # prepare decoder-inputs
        targets, target_masks  = self.prepare_targets(batch, targets, distillation, target_masks)  # prepare decoder-targets
        sources, source_masks  = self.prepare_sources(batch, source_masks)
        encoding = self.encoding(sources, source_masks)
        return inputs, input_masks, targets, target_masks, sources, source_masks, encoding, inputs.size(0)

    def forward(self, encoding, encoder_masks, decoder_inputs, decoder_masks,
                decoding=False, beam=1, alpha=0.6, return_probs=False, positions=None, feedback=None):

        if (return_probs and decoding) or (not decoding):
            out = self.decoder(decoder_inputs, encoding, encoder_masks, decoder_masks)

        if decoding:
            if beam == 1:  # greedy decoding
                output = self.decoder.greedy(encoding, encoder_masks, decoder_masks, feedback=feedback)
            else:
                output = self.decoder.beam_search(encoding, encoder_masks, decoder_masks, beam, alpha)
```

Salesforce Einstein

Batching in Deep Learning

And padding and masking aren't enough to make even basic language-native control flow work in general.

```
# x is a batch of scalars
while x > 0:
    x = x - 1
return x
```

```
# shift-reduce parsing
for transition in transitions:
    if transition == SHIFT:
        stack.append(buffer.pop())
    elif transition == REDUCE:
        stack.append(compose(stack.pop(),
                             stack.pop()))
```

Salesforce Einstein

Batching in Deep Learning

While many of these examples are motivated by natural language processing, network structures with example-dependent control flow appear in other fields too:

Graph convolutions (biochemistry)
Neural module networks (visual QA)
RL architectures for games, knowledge graphs, and databases

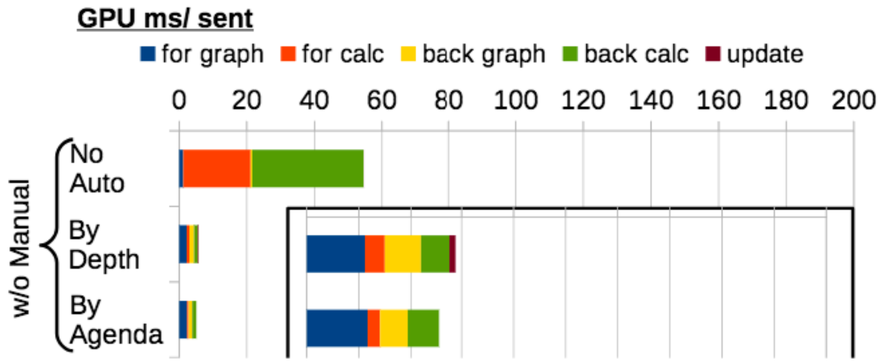
Automatic Batching: TensorFlow Fold

- A functional subset of TensorFlow embedded into Python as a domain-specific language
  - Essentially another language that programmers have to learn
- The network structure is allowed to depend on the structural type of the input data but not on runtime values.
  - Only includes LISPy control flow operators, not `while` and `if`.

Salesforce Einstein

Automatic Batching: DyNet autobatch

- Lazily constructs computation graphs for each example before applying batching/vectorization as a global graph optimization
  - The graph structure still can't depend on runtime values
- Modern GPUs are so fast that the per-example graph construction plus the global optimization takes longer than graph execution



From "On-the-fly Operation Batching in Dynamic Computation Graphs," Neubig et al. NIPS 2017
GPU is NVIDIA Tesla K80

Automatic Batching: Matchbox

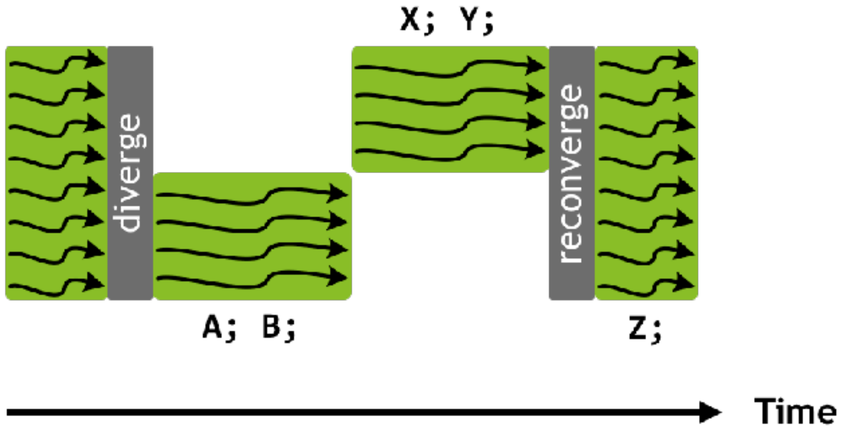
- Well-written manual batching (as found in packages like AllenNLP) already covers non-control-flow cases well, so let's automate it!

```
# We replace masked values with something really negative here, so they don't affect the
# max below.
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                               question_mask.unsqueeze(1),
                                               -1e7)
# Shape: (batch_size, passage_length)
question_passage_similarity = masked_similarity.max(dim=-1)[0].squeeze(-1)
# Shape: (batch_size, passage_length)
question_passage_attention = util.masked_softmax(question_passage_similarity, passage_mask)
# Shape: (batch_size, encoding_dim)
```

Automatic Batching: Matchbox

- Instead of treating batching as a generic compiler problem because we want to support generic control flow, let's take advantage of the SIMT-like structure of deep learning code.
  - Computation graphs for each example are almost always more similar than they are different

```
if (threadIdx.x < 4) {
    A;
    B;
} else {
    X;
    Y;
}
Z;
```



From NVIDIA CUDA developer material

How Matchbox Works

- The **`MaskedBatch`** type behaves like a PyTorch **`Tensor`** but represents a batch of examples that may vary in size along a specified subset of their dimensions (**dynamic dimensions** vs static ones).

- This is accomplished by storing a mask which is automatically propagated by PyTorch operations (methods and neural network layers)

CATEGORY	EXAMPLE	DATA ARRAY	MASK ARRAY	DIM. SPEC.
POINTWISE UNARY	tanh	$f(x)$	m_x	d_x
POINTWISE BINARY	$+$	$f(x, y)$	$m_x \wedge m_y$	$d_x \vee d_y$
REDUCTION	mean, $\neg d_x[\text{dim}]$	$f(x * m_x, \text{dim})$	$\text{sqz}(m_x, \text{dim})$	d_x
SOFTMAX	$d_x[\text{dim}]$	$f(x * m_x, \text{dim}) * m_x / \text{sum}(x, \text{dim})$	$\text{sqz}(m_x, \text{dim})$	$\text{flip}(d_x, \text{dim})$
MATRIX MULTIPLY	x, y 3D	$f(x * m_x, y * m_y)$	$f(\text{sqz}(m_x, 2), \text{sqz}(m_y, 1))$	$(d_x[0], d_y[1])$

Salesforce Einstein

How Matchbox Works

- The **`MaskedBatch`** type behaves like a PyTorch **`Tensor`** but represents a batch of examples that may vary in size along a specified subset of their dimensions (**dynamic dimensions** vs static ones).

- This is accomplished by storing a mask which is automatically propagated by PyTorch operations (methods and neural network layers)

```
def _elementwise_unary(fn):
    def inner(batch, *args, **kwargs):
        if not isinstance(batch, MaskedBatch):
            return fn(batch, *args, **kwargs)
        data = fn(batch.data, *args, **kwargs)
        mask = batch.mask.type_as(data)
        dims = batch.dims
        return MaskedBatch(data, mask, dims)
    return inner
```

```
MaskedBatch.log = log = _elementwise_unary(TENSOR_TYPE.log)
MaskedBatch.sqrt = sqrt = _elementwise_unary(TENSOR_TYPE.sqrt)
MaskedBatch.sin = sin = _elementwise_unary(TENSOR_TYPE.sin)
MaskedBatch.cos = cos = _elementwise_unary(TENSOR_TYPE.cos)
MaskedBatch.tan = tan = _elementwise_unary(TENSOR_TYPE.tan)

MaskedBatch.relu = relu = _elementwise_unary(F.relu)
MaskedBatch.tanh = tanh = _elementwise_unary(F.tanh)
MaskedBatch.sigmoid = sigmoid = _elementwise_unary(F.sigmoid)
```

Salesforce Einstein

How Matchbox Works

- Control flow is vectorized using SIMT-like execution masking and data synchronization primitives added by the `@batch` decorator

```
class BiRNN(nn.Module):
    def __init__(self, size):
        super().__init__()
        self.fwd = nn.RNNCell(size, size)
        self.bwd = nn.RNNCell(size, size)

    @batch
    def forward(self, x):
        h = h0 = x.batch_zeros(x.size(-1))
        fwd, bwd = [], []
        for xt in x.unbind(1):
            h = self.fwd(xt, h)
            fwd.append(h)
        fwd = F.stack(fwd, 1)
        h = h0
        for xt in reversed(x.unbind(1)):
            h = self.bwd(xt, h)
            bwd.append(h)
        bwd = F.stack(reversed(bwd), 1)
        return F.cat((fwd, bwd), 2)
```

```
def forward(self, x):
    h = h0 = x.batch_zeros(x.size(-1))
    fwd, bwd = [], []
    for xt in x.unbind(1):
        h = h._update(self.fwd(xt, h))
        fwd.append(h)
        h = h._synchronize()
    fwd = F.stack(fwd, 1)
    h = h0
    for xt in reversed(x.unbind(1)):
        h = h._update(self.bwd(xt, h))
        bwd.append(h)
        h = h._synchronize()
    bwd = F.stack(reversed(bwd), 1)
    return F.cat((fwd, bwd), 2)
```

How Matchbox Works

- The package also provides some additional convenience methods for example-level programming; these are implemented both for batch and tensor objects, because **all code written for Matchbox also works with plain Tensors and batch size one**.

- This means testing Matchbox correctness is straightforward: users can compare results from a loop over several examples with batch size one against results from the same examples in a Matchbox batch.

    - Similar to gradient checking tools, the provided `mb_test` wrapper does this automatically.

Example: Transformer

Google Brain's Transformer, from "Attention Is All You Need," is a machine translation model based on self-attention.

```
class Attention(nn.Module):
   def __init__(self, dk, drop, causal):
      super().__init__()
      self.scale = math.sqrt(dk)
      self.drop = nn.Dropout(drop)
      self.causal = causal

   def forward(self, q, k, v):
      a = q @ k.transpose(1, 2)
      if self.causal:
         a = a.causal_mask(2, 1)
      return self.drop((a/self.scale)
                       .softmax()) @ v
```

```
class MultiHead(nn.Module):
   def __init__(self, attn, dk, dv, N):
      super().__init__()
      self.attn = attn
      self.wq = nn.Linear(dk, dk)
      self.wk = nn.Linear(dk, dk)
      self.wv = nn.Linear(dv, dv)
      self.wo = nn.Linear(dv, dk)
      self.N = N

   def forward(self, q, k, v):
      q = self.wq(q)
      k = self.wk(k)
      v = self.wv(v)
      # B,T,D -> B,T,D/N,N -> B*N,T,D/N
      q, k, v = (x.split_dim(-1, self.N)
                  .join_dims(0, -1)
                  for x in (q, k, v))
      o = self.attn(q, k, v)
      # B*N,T,D/N -> B,N,T,D/N -> B,T,D
      o = (o.split_dim(0, self.N)
           .join_dims(-1, 1))
      return self.wo(o)
```

Salesforce Einstein

Example: Novel Research Model

A snippet from an in-progress research project that was initially written at example level and uses native control flow

```
@batch
def calc_n_expansions(self, n_leaves):
    if self.n_expansions_mode == 'sparse':
        return n_leaves - 1

    else:
        if self.n_expansions_mode == 'dense':
            parent_conn_usage = 1.0
        else:
            parent_conn_usage = 0.5  # 'medium'

        c_per_parent = 1 + parent_conn_usage * (
            self.n_relations - 1)

        unconnected = n_leaves.float()
        expansions = unconnected.new_zeros(
            unconnected.size(0))
        while unconnected > 1:
            unconnected /= c_per_parent
            expansions += unconnected.ceil()

        expansions = expansions.clamp(1).long()
        return expansions
```

Salesforce Einstein

Similar Efforts

- The deep learning framework CNTK allows tensors to have a single optional dynamic dimension, which usually represents time. Kernel generators like TVM and Tensor Comprehensions also have a concept of static vs dynamic dimensions.

- Python compilers and tracers like Myia, JAX, Tangent, and the PyTorch JIT often contain similar control flow transformation passes.

Some Limitations

- Matchbox only works on code that uses native PyTorch operators (that means no Python scalars for any quantities that vary between examples and no NumPy ops)
- Control flow support is limited (e.g., no `return` from within a `for`)
- There's a long tail of less-common operations that haven't been implemented (plus bigger gaps, like convolutions)
- It's possible to port existing batched code to Matchbox, but the primary goal is to enable writing new models natively at example-level

Overhead and JIT Support

- Matchbox adds a small runtime overhead, largely because it's implemented in Python

Mode	Time per iteration (s)	
	K40	V100
Unbatched	2.704	1.925
Batched without masking	0.479	0.108
Matchbox	0.516	0.150

- Because Matchbox operations are all implemented in terms of native PyTorch operations, we can rely on the in-progress PyTorch tracer and compiler to lift calls and control flow out of Python

Future Work

- Adding `MaskedBatch` support for more operations
- A separate `PackedBatch` type that can pack its data tensor along its batch dimension and one dynamic dimension and stores a separate tensor of offsets.
    - This type will be natively compatible with cuDNN RNNs and save memory relative to `MaskedBatch`, but will be slower for some operations.

Julia

- Both of the main features of Matchbox rely on programming language features (dispatch and code transformation) that are fairly inconvenient in Python
    - We rely heavily on “single-dispatch” (method calls) even when it might not make sense
- A prototype for a Matchbox in a language that natively supports these things (Julia) is at github.com/jekbradbury/Minibatch.jl

James Bradbury
@jekbradbury

.@JeffDean at #SysML: having to work directly with batching in ML models "kind of makes my head hurt sometimes"

3:30 PM - 16 Feb 2018

github.com/salesforce/matchbox

Salesforce Einstein