网络出版地址：http://kns.cnki.net/kcms/detail/11.2442.N.20180822.2013.015.html

北京大学学报(自然科学版)
Acta Scientiarum Naturalium Universitatis Pekinensis
doi: 10.13209/j.0479-8023.2018.065

N3LDG：一种轻量级自然语言处理深度学习库

王潜升　余南　张梅山　韩子嘉　付国宏†

黑龙江大学计算机科学技术学院 哈尔滨 150080; † 通信作者, E-mail: ghfu@hotmail.com

摘要　提出一种用于自然语言处理的轻量级深度学习库 N3LDG。N3LDG 可以支持动态地构建计算图，并能自动批量化执行计算图。实验显示，当训练卷积神经网络、双向 LSTM 和树结构 LSTM 时, N3LDG 都能高效地构建与执行计算图。当使用 CPU 训练上述模型时, N3LDG 的训练速度优于 PyTorch。当使用 GPU 训练卷积神经网络和树结构 LSTM 模型时, N3LDG 的训练速度优于 PyTorch。

关键词　深度学习库; 自然语言处理; 轻量级; CUDA

N3LDG: A Lightweight Neural Network Library for Natural Language Processing

WANG Qiansheng, YU Nan, ZHANG Meishan, HAN Zijia, FU Guohong†
School of Computer Science and Technology, Heilongjiang University, Harbin 150080;
† Corresponding author, E-mail: ghfu@hotmail.com

Abstract　The authors propose a neural network library N3LDG for natural language processing. N3LDG supports constructing computation graphs dynamically, and organizing executions into batches automatically. Experiments show that N3LDG can efficiently construct and execute computation graphs when training CNN, Bi-LSTM, and Tree-LSTM. When using CPU to train above models, the training speed of N3LDG is better than that of PyTorch. When using GPU to train CNN and Tree-LSTM, N3LDG is better than PyTorch.

Key words　deep learning library; NLP; lightweight; CUDA

近年来，深度学习方法在自然语言处理领域很多任务中的性能超越了传统的统计机器学习方法，得到了广泛应用[1]。在训练阶段，深度学习模型的执行步骤包括前向传播、反向传播和更新参数等，深度学习库能方便地执行这些步骤。Theano[2]、CNTK[3]、Caffe[4]、TensorFlow[5]和PyTorch[6]等已得到广泛应用[7][8][9][10]。

Theano[2]、CNTK[3]、Caffe[4]和 TensorFlow[5]在训练前静态定义计算图，训练时对于所有实例执行同一个计算图。而在自然语言处理任务中，构建适应所有实例的计算图存在额外的困难，体现在以下两方面。

1) 各实例的长度不一致。补零可使各实例长度一致，然而补零操作可能影响计算结果。为了避免这个影响，需对计算结果做裁剪。

2) 实例含有结构化信息，如句法结构。有时我们希望基于这些结构化信息动态构建计算图，比如在基于句法的递归神经网络中，不同的句子实例有着不同的句法结构，也就对应于不同的计算图。

PyTorch 等深度学习库则根据不同的实例，动态构建不同的计算图。为了利用多核 CPU 或 GPU 加速计算, PyTorch 要求使用者把能批量化计算的数据手动合并为张量。比如在计算机视觉任务中，使用者须将多个图像实例，合并为一个张量后作为模型的输入。

国家自然科学基金(61672211, 61602160)和黑龙江省自然科学基金(F2016036)资助
收稿日期: 2018–04–21; 修回日期: 2018–08–15; 网络出版时间: 2018-08-22 20:23:39

而在自然语言处理任务中，手动批量化存在以下额外的困难：1) 各句子实例须在补零后才能合并为张量；2) 在树结构模型如递归神经网络中，须分析计算图执行步骤后，将同一执行步骤中，处于不同实例上的计算过程批量化。

Looks 等[11]提出一种自动批量化方法，这种方法允许深度学习库构建完整个计算图后，自动发现当前可执行的同类型计算过程，并将它们批量化执行。为方便自然语言处理领域的研究者使用，我们也实现了动态计算图和自动批量化。如同多数深度学习库[2][3][4][5][6]一样，我们还实现了自动微分。N3LDG 将向量视为计算的对象，将卷积、池化等视为基于向量的各种操作，而在自然语言处理任务中，深度学习模型的输入通常是词向量或拼接了其他特征的向量，这样 N3LDG 满足了自然语言处理任务的要求。为提高执行速度，N3LDG 使用 C++实现。相比其他深度学习库，N3LDG 更为轻量级，只需在项目中包含头文件即可使用。在 Apache 2.0 协议下，N3LDG 在 https://github.com/zhangmeishan/N3LDG 发布。

1　相关工作

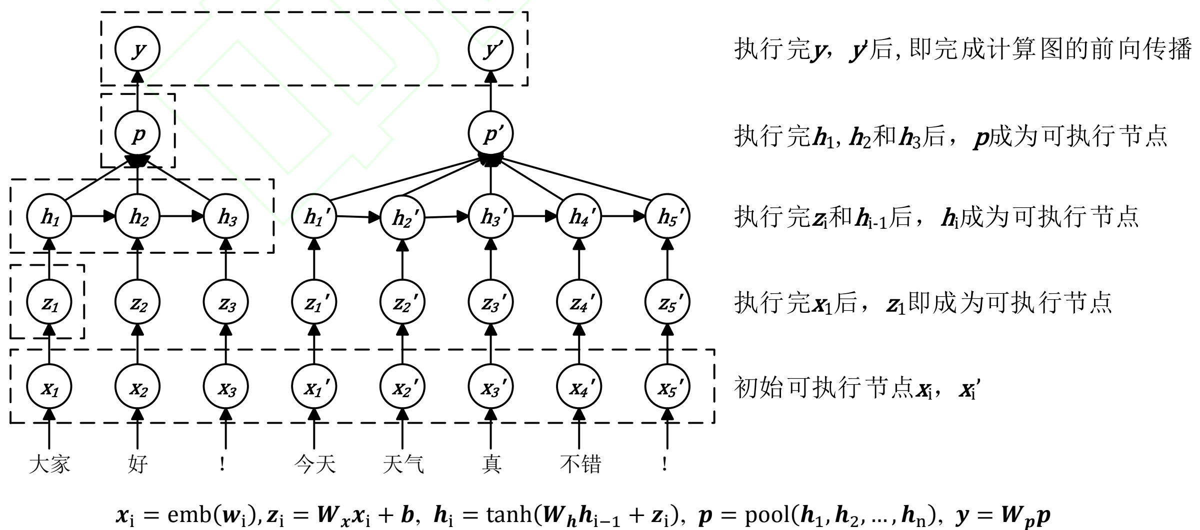
近年来，出现了很多通用深度学习库。Zhang 等[12]提出一种自然语言处理深度学习库 LibN3L，实现了深度学习模型中的常见操作，然而该库不支持自动批量化。针对深度学习模型的计算图的自动批量化研究并不多。Looks 等[11]首先提出基于节点在计算图中的深度的自动批量化方法后，Neubig 等[13]讨论了这个方法在处理 RNN 模型时，难以充分批量化。为缓解这个问题，他们提出一种将同类型节点在计算图中的平均深度作为启发式规则的方法，并应用在他们的深度学习库 DyNet[14]中。由于 RNN 模型在自然语言处理任务中很常用，为了能高效训练 RNN 模型，我们仿照 Neubig 等[13]的方法。

多数深度学习库能利用 GPU 加速训练模型[2][3][4][5][6][14]。Chetlur 等[15]提出了 cuDNN 库，高效地实现了深度学习中的各基本操作。为了高效地分配显存，DyNet[14]在库初始化时创建了 3 个显存块，其中一个显存块用于在前向传播中使用，另一个用于在反向传播中使用，最后一个用于存储参数和相关的梯度，这样，可通过指针的加减运算实现分配和释放显存的操作。我们通过显存池来高效地分配显存，这种做法不要求使用者预估显存占用，而是在需要时动态地向系统申请新的显存块。

2　计算图

2.1　计算图的引入

对于一个简单的线性分类器$\boldsymbol{y}=\mathrm{W}\boldsymbol{x}$，只需以 $\boldsymbol{x}$ 为自变量作线性变换，便可得到分类结果。为说明计算图的优点，我们引入更复杂的模型，图 1 描述了一种循环神经网络(RNN)模型。



$$\boldsymbol{x}_\mathrm{i}=\mathrm{emb}(\boldsymbol{w}_\mathrm{i}),\ \boldsymbol{z}_\mathrm{i}=\boldsymbol{W}_x\boldsymbol{x}_\mathrm{i}+\boldsymbol{b},\ \boldsymbol{h}_\mathrm{i}=\tanh(\boldsymbol{W}_h\boldsymbol{h}_{\mathrm{i}-1}+\boldsymbol{z}_\mathrm{i}),\ \boldsymbol{p}=\mathrm{pool}(\boldsymbol{h}_1,\boldsymbol{h}_2,\ldots,\boldsymbol{h}_\mathrm{n}),\ \boldsymbol{y}=\boldsymbol{W}_p\boldsymbol{p}$$

图 1　一种 RNN 模型

Fig. 1　An RNN model

该模型可表示为$\boldsymbol{y}=\mathrm{f}(\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n})$，我们难以直接表示出 f, 因此将 f 分解为多个简单的计算步骤, 每步计算结果存储在一个中间向量中。当把向量视作图的节点, 向量之间形成有向边, 分解后的各计算步骤和向量构成计算图 G。

为实现计算图, 首先定义 Node 类作为计算图节点。以图 1 中$\boldsymbol{h}_\mathrm{i}=\tanh(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{h}_{\mathrm{i}-1}+\boldsymbol{z}_\mathrm{i})$为例, 为了计算 $\boldsymbol{h}_\mathrm{i}$, Node 类需包含以下信息: 1) 前向传播的计算方法; 2) 本节点向量($\boldsymbol{h}_\mathrm{i}$); 3) 各输入向量($\boldsymbol{h}_{\mathrm{i}-1}$、$\boldsymbol{z}_\mathrm{i}$); 4) 参数($\boldsymbol{W}_{\boldsymbol{h}}$)。当给定各输入向量$\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n}$, 某节点即可执行前向传播过程, 求得本节点向量$\boldsymbol{y}=\mathrm{f}(\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n})$，称该节点为可执行节点。$\boldsymbol{y}$ 又可作为其子节点的输入向量, 使子节点成为可执行节点。若某节点不含输入向量(图 1 $\boldsymbol{x}_\mathrm{i}$所在节点), 则该节点成为计算图 G 的初始可执行节点。这样以初始可执行节点为始, 重复执行前向传播过程, 直至计算图中所有节点都被执行, 即完成模型的前向传播过程。图 1 描述了这个过程。

为执行反向传播, Node 类还须包含以下信息: 1) 反向传播的计算方法; 2) 损失函数 L 对 $\boldsymbol{y}$ 的导数$\frac{\mathrm{dL}}{d\boldsymbol{y}}$; 3) L 对各输入向量的导数$\frac{\mathrm{dL}}{d\boldsymbol{x}_1},\frac{\mathrm{dL}}{d\boldsymbol{x}_2},\dots,\frac{\mathrm{dL}}{d\boldsymbol{x}_\mathrm{n}}$; 4) L 对各参数的导数$\frac{\mathrm{dL}}{d\mathbf{W}_1},\frac{\mathrm{dL}}{d\boldsymbol{W}_2},\dots,\frac{\mathrm{dL}}{d\boldsymbol{W}_\mathrm{n}}$。我们在 Node 类中定义前向传播和反向传播的接口, 在其各个子类中实现这两个接口。我们实现了常用的节点类型, 包括 tanh, concat 和线性变换等, 列举在表 1。使用者也可定义新的节点类型, 自己实现前向传播和反向传播。

2.2 自动批量化

计算图中往往有多个可执行节点。为提高执行速度, 需批量化执行同类型的计算过程。具体而言, 有两类计算过程可批量化执行: 1) 共享参数的同类型计算, 如$\boldsymbol{y}_1=\boldsymbol{W}\boldsymbol{x}_1+\boldsymbol{b}$和$\boldsymbol{y}_2=\boldsymbol{W}\boldsymbol{x}_2+\boldsymbol{b}$, 2) 不含参数矩阵的同类型计算, 如$\boldsymbol{y}_1=\tanh(\boldsymbol{x}_1)$和$\boldsymbol{y}_2=\tanh(\boldsymbol{x}_2)$。

N3LDG 自动发现当前可执行节点, 并批量化执行同类型的计算过程。当这些节点被执行完成后, 即从计算图中移除, 此时可得新的可执行节点集合。这样总能得到当前可执行节点的集合, 直到计算图执行完毕。以图 1 中 RNN 模型为例, 执行步骤如下。

(1) $[\boldsymbol{x}_1\ \boldsymbol{x}_2\ \cdots\ \boldsymbol{x}_5']=$ $\left[\mathrm{emb}(\text{大家})\ \mathrm{emb}(\text{好})\ \cdots\ \mathrm{emb}(\text{！})\right]$。

(2) $[\boldsymbol{z}_1\ \boldsymbol{z}_2\ \cdots\ \boldsymbol{z}_5']=W_{\boldsymbol{x}}[\boldsymbol{x}_1\ \boldsymbol{x}_2\ \cdots\ \boldsymbol{x}_5']+$ $[\boldsymbol{b}\ \boldsymbol{b}\ \cdots\ \boldsymbol{b}]$。

(3) $[\boldsymbol{h}_1\ \boldsymbol{h}_1']=\tanh([\boldsymbol{z}_1\ \boldsymbol{z}_1'])$。

(4) $[\boldsymbol{h}_2\ \boldsymbol{h}_2']=\tanh(\boldsymbol{W}_{\boldsymbol{h}}[\boldsymbol{h}_1\ \boldsymbol{h}_1']+[\boldsymbol{z}_2\ \boldsymbol{z}_2'])$。

(5) $[\boldsymbol{h}_3\ \boldsymbol{h}_3']=\tanh(\boldsymbol{W}_{\boldsymbol{h}}[\boldsymbol{h}_2\ \boldsymbol{h}_2']+[\boldsymbol{z}_3\ \boldsymbol{z}_3'])$。

(6) $\boldsymbol{h}_4'=\tanh(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{h}_3'+\boldsymbol{z}_4')$。

(7) $\boldsymbol{h}_5'=\tanh(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{h}_4'+\boldsymbol{z}_5')$。

(8) $[\boldsymbol{p}\ \boldsymbol{p}']=$ $[\mathrm{pool}(\boldsymbol{h}_1,\boldsymbol{h}_2,\boldsymbol{h}_3)\ \mathrm{pool}(\boldsymbol{h}_1',\boldsymbol{h}_2',\dots,\boldsymbol{h}_5')]$。

(9) $[\boldsymbol{y}\ \boldsymbol{y}']=\boldsymbol{W}_{\boldsymbol{p}}[\boldsymbol{p}\ \boldsymbol{p}']$。

当遇到同时有多种计算过程可执行时, 比如当执行$[\boldsymbol{h}_3\ \boldsymbol{h}_3']=\tanh(\boldsymbol{W}_{\boldsymbol{h}}[\boldsymbol{h}_2\ \boldsymbol{h}_2']+[\boldsymbol{z}_3\ \boldsymbol{z}_3'])$后, 此时既可执行$\boldsymbol{h}_4'=\tanh(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{h}_3'+\boldsymbol{z}_4')$, 也可执行$\boldsymbol{p}=\mathrm{pool}(\boldsymbol{h}_1,\boldsymbol{h}_2,\boldsymbol{h}_3)$, 为了批量执行池化操作, 应先执行$\boldsymbol{h}_4'=\tanh(\boldsymbol{W}_{\boldsymbol{h}}\boldsymbol{h}_3'+\boldsymbol{z}_4')$。我们参照 Neubig 等[13]的方法, 对某计算类型的节点求平均深度, 以此作为作为启发函数, 优先执行平均深度更小的计算过程。

表 1　N3LDG 中的常用节点类型

Table 1　Commonly used node types in N3LDG

节点类型	计算过程
LookupNode	$\boldsymbol{y}=\mathrm{emb}(w)$
ConcatNode	$\boldsymbol{y}=\boldsymbol{x}_1\oplus\boldsymbol{x}_2\dots\oplus\boldsymbol{x}_\mathrm{n}$
LinearNode	$\boldsymbol{y}=\boldsymbol{W}\boldsymbol{x}+\boldsymbol{b}$
TanhNode	$\boldsymbol{y}=\tanh(\boldsymbol{x})$
DropoutNode	$\boldsymbol{y}=\mathrm{dropout}(\boldsymbol{x})$
PAddNode	$\boldsymbol{y}=\boldsymbol{x}_1+\boldsymbol{x}_2+\cdots+\boldsymbol{x}_\mathrm{n}$
PMultiNode	$\boldsymbol{y}=\boldsymbol{x}_1\odot\boldsymbol{x}_2$
MaxPoolNode	$\mathrm{maxpool}(\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n})$
MinPoolNode	$\mathrm{minpool}(\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n})$
AvgPoolNode	$\mathrm{avgpool}(\boldsymbol{x}_1,\boldsymbol{x}_2,\dots,\boldsymbol{x}_\mathrm{n})$
AttentionSoftMaxNode	$[\boldsymbol{x}_1\ \boldsymbol{x}_2\ \dots\ \boldsymbol{x}_\mathrm{n}]\mathrm{softmax}(\boldsymbol{w})$

3 CPU 计算

Eigen[14]是通用的C++线性代数计算库, 我们使用 Eigen 实现 CPU 上的线性代数计算。由于 CPU 能高效地处理在内存中连续存放的向量, 所以 N3LDG 对常用的计算过程作了优化。具体的步骤如下: 1) 计算前, 将各节点中的输入向量合并为一个矩阵, 将矩阵与多个向量的乘法运算转换为矩阵与矩阵的乘法运算; 2) 执行矩阵和矩阵的乘法运算, 得到结果矩阵; 3) 将该结果矩阵拆分后, 赋值给各节点的向量。

以 $\boldsymbol{y_1} = \tanh(\mathrm{W}\,\boldsymbol{x_1} + \boldsymbol{b}), \boldsymbol{y_2} = \tanh(\mathrm{W}\,\boldsymbol{x_2} + \boldsymbol{b})\ldots\boldsymbol{y_n} = \tanh(\mathrm{W}\,\boldsymbol{x_n} + \boldsymbol{b})$为例, 首先将 $\boldsymbol{x_1}$, $\boldsymbol{x_2}\ldots\boldsymbol{x_n}$ 合并为矩阵$[\boldsymbol{x_1}\,\boldsymbol{x_2} \ldots \boldsymbol{x_n}]$, 记为 $\boldsymbol{X}$, 将同一个向量 $\boldsymbol{b}$ 扩展为 n 列矩阵$[\boldsymbol{b}\,\boldsymbol{b} \ldots \boldsymbol{b}]$, 记为 B。将各向量拷贝至连续的内存区域中, 然后执行 $\mathrm{Y} = \tanh(\mathrm{WX} + \mathrm{B})$, 计算完成后, 将矩阵 Y 拆分拷贝至各节点的向量。

4 GPU 计算

cuBLAS 是英伟达发布的 CUDA 线性代数计算库, 我们使用 cuBLAS 实现 GPU 上的线性代数计算, 并编写 kernel 函数实现其余计算过程。为充分利用 GPU 的并行计算能力, 我们并行执行了所有批量化之后的计算过程。我们的实现不依赖于 cuDNN[15], 使用者无需安装 cuDNN。

我们发现 GPU 中有两类操作存在性能瓶颈: 1) 显存分配与释放; 2) 显存和内存间的 I/O。当动态构建计算图时, 参与前向传播和反向传播计算过程的各向量地址也随之动态变化, 计算前须将这些信息传输到显存, 这频繁涉及上述两类操作。我们通过以下方法来缓解了性能瓶颈。

4.1 显存分配与释放

在实验中我们测量了显存的分配与释放时间, 发现它们占总训练时间相当大的比例, 成为性能瓶颈。我们通过专用模块(显存池), 持有并管理空闲的显存块, 当不持有合适的空闲块时才向系统申请显存块, 从而减少向系统分配与释放显存的次数。英伟达实现了显存池库 cnmem, 在 https://github.com/NVIDIA/cnmem 发布。Knowlton 等[17]提出伙伴系统, 用于快速地分配存储空间。受伙伴系统启发, 我们也实现了显存池。

4.2 I/O

对于同样大小的数据, 只调用一次库函数将其传输至显存, 显著地快于分成多次传输[18]。因此对于需传输至显存的多个数据, 我们在内存中将其连续存放后, 再调用一次库函数传至显存。比如, 批量化执行 $\boldsymbol{y_1} = \tanh(\mathbf{x_1}), \boldsymbol{y_2} = \tanh(\mathbf{x_2}) \ldots \boldsymbol{y_n} = \tanh(\mathbf{x_n})$ 时, 需将 $\boldsymbol{x_1}$, $\boldsymbol{x_2}\ldots\boldsymbol{x_n}$ 的地址传输至显存, 我们在内存中连续存放 $\boldsymbol{x_1}$, $\boldsymbol{x_2}\ldots\boldsymbol{x_n}$ 的地址后, 调用一次库函数将这些地址传输至显存。相比于调用多次库函数分别传输它们的地址, 我们的方法显著减少了显存与内存间的 I/O 次数。

5 实验

通过一个 5 分类情感分类任务, 在 3 个模型上做基准测试: 1) 卷积神经网络(CNN); 2) 双向长短时记忆网络(Bi-LSTM); 3) 树结构长短时记忆网络(Tree-LSTM)[19]。以上模型的词向量和隐层的维度都设置为 200。训练数据包括 8544 个句子实例, 共 163563 个词(包括标点符号), 测试代码在 https://github.com/chncwang/n3ldg-benchmark 发布。我们记录训练一轮 epoch 的时长, 包括: 1) 构建计算图; 2) 规划执行步骤; 3) 前向传播; 4) 反向传播; 5) 更新参数。实验中 CPU 型号是 Intel(R) Core(TM) i7-6800K CPU @ 3.40 GHz, GPU 型号是 GeForce GTX 1080 Ti。我们还用 PyTorch 实现一致的模型结构, 并在 3 个模型上实现手动批量化, 以便与 N3LDG 对比训练速度。在 N3LDG 中, 我们未对 LSTM 作特别的优化, 为公平对比, 我们用 PyTorch 以同样方式实现 LSTM。

首先在单线程 CPU 上, 我们对 N3LDG 和 PyTorch 作了基准测试, 测试结果见表 2。

表 2 显示, 在所有设置下, N3LDG 在单线程 CPU 上的训练速度高于 PyTorch。当训练 CNN 时, N3LDG 训练速度达到 PyTorch 的 9.40~42.47 倍, 训练 Bi-LSTM 时达到 PyTorch 的 4.43~9.71 倍, 训练 Tree-LSTM 时达到 PyTorch 的 1.28~3.10 倍。这表明我们构建计算图、自动批量化和 CPU 计算过程是高效的。

我们在 GPU 上对 N3LDG、不使用 cuDNN 的 PyTorch(称之为 PyTorch CUDA)以及使用 cuDNN 的 PyTorch(PyTorch cuDNN)做了基准测试, 测试结果见表 3。

表 3 显示, 在训练 CNN 和 Tree-LSTM 时, N3LDG 在 GPU 上的训练速度高于 PyTorch CUDA 和 PyTorch cuDNN。在训练 CNN 时, N3LDG 训练速度达到 PyTorch CUDA 的 3.40~18.38 倍, PyTorch cuDNN 的 3.10~8.74 倍, 训练 Tree-LSTM 时达到 PyTorch CUDA 的 1.78~3.03 倍, PyTorch cuDNN 的 1.67~2.79 倍。训练 Bi-LSTM 模型时, N3LDG 在较大的 mini-batch 下有优势。

当 mini-batch=1 时, N3LDG 训练速度低于 PyTorch, 是 PyTorch CUDA 的 77.46%, PyTorch cuDNN 的 80.18%。当 mini-batch=16 时, N3LDG 的训练速度与 PyTorch 几乎相同。当 mini-batch=256 时, N3LDG 的训练速度达到 PyTorch CUDA 的 1.71 倍, PyTorch cuDNN 的 1.77 倍。总体而言, 我们认为构建计算图、自动批量化、GPU 计算过程是高效的。

为了分析自动批量化对训练速度的影响, 以 Bi-LSTM (MB=256)为例, 分别在单线程 CPU 和 GPU 上测试了是否做自动批量化时, 各步骤的时长。实验结果见图 2。

图 2 显示, 自动批量化显著提升了训练速度。在单线程 CPU 上的提升 4.76 倍, 在 GPU 上提升 52.27 倍。提速主要来自前向传播和反向传播。

我们猜测在单线程CPU上提速的部分原因在于合并了矩阵与向量的乘法, 比如将$\mathbf{y_1} = \mathbf{W}\,\boldsymbol{x_1}$和$\boldsymbol{y_2} = \mathbf{W}\,\boldsymbol{x_2}$转换为为$[\boldsymbol{y_1} \quad \boldsymbol{y_2}] = \mathbf{W}\,[\boldsymbol{x_1} \quad \mathbf{x_2}]$后, 计算更快。我们还对比了在相同设置下, 一轮 epoch 中矩阵乘法的总执行时间、执行次数及平均执行时间, 见表 4。

表 2 单线程 CPU 上 N3LDG 和 PyTorch 的基准测试

Table 2 Benchmarks of N3LDG and PyTorch on single thread CPU

模型	时间/s	
	N3LDG	PyTorch
CNN (MB=1)	**14.76**	626.96
CNN (MB=16)	**7.56**	91.61
CNN (MB=256)	**6.92**	65.03
Bi-LSTM (MB=1)	**116.88**	1135.03
Bi-LSTM (MB=16)	**35.46**	157.12
Bi-LSTM (MB=256)	**23.69**	108.03
Tree-LSTM (MB=1)	**34.67**	107.61
Tree-LSTM (MB=16)	**20.39**	54.56
Tree-LSTM (MB=256)	**18.46**	23.67

注: MB 表示 mini-batch 大小, 下同。

表 3 GPU 上 N3LDG、PyTorch CUDA 和 PyTorch cuDNN 的基准测试

Table 3 Benchmarks of N3LDG, PyTorch CUDA and PyTorch cuDNN on GPU

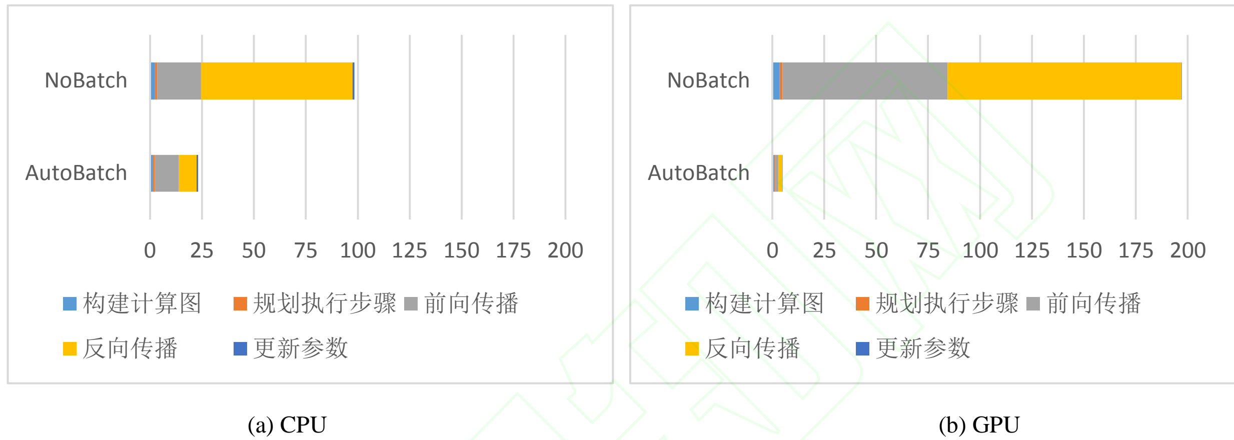
模型	时间/s		
	N3LDG	PyTorch CUDA	PyTorch cuDNN
CNN (MB=1)	**6.71**	22.82	20.79
CNN (MB=16)	**0.80**	8.81	5.16
CNN (MB=256)	**0.47**	8.64	4.11
Bi-LSTM (MB=1)	183.27	**141.97**	146.95
Bi-LSTM (MB=16)	24.94	**24.30**	24.78
Bi-LSTM (MB=256)	**4.75**	8.116	8.414
Tree-LSTM (MB=1)	**49.32**	112.94	91.72
Tree-LSTM (MB=16)	**13.83**	24.59	23.12
Tree-LSTM (MB=256)	**5.35**	16.20	14.93

表 4 显示自动批量化显著提升了单线程 CPU 矩阵乘法的执行速度，提升了 9.28 倍。与不作批量化相比，尽管自动批量化时平均执行时间更长，但执行次数仅为 0.98%。

为分析自动批量化对 CUDA 核函数执行速度的影响，我们对比了在相同设置下，一轮 epoch 中核函数的总执行时间、执行次数及平均执行时间，见表 5。

表 5 显示，自动批量化显著提升了 CUDA 核函数的执行速度，达到 70.52 倍。与不做批量化相比，执行次数仅为 1.28%，值得注意的是，平均执行时间只是不作批量化时的 1.11 倍，这表明自动批量化充分利用了 GPU 的并行计算能力。

为测试显存池的有效性，我们在训练 Bi-LSTM 时，分别统计了一轮 epoch 中，不使用显存池、使用 cnmem 以及使用我们的显存池时的训练时间以及分配与释放显存的时长，实验结果见图 3。



(a) CPU　　　　(b) GPU

图 2　做自动批量化与否时各训练阶段时长

Fig. 2　Time in per training stage when auto-batch enabled or not

表 4　单线程 CPU 矩阵乘法的总执行时间、执行次数和平均执行时间

Table 4　Total execution duration, times and average duration of matrix multiplication

方法	总时间/s	执行次数	平均执行时间/ns
No Batch	80.93	7774128	10.41
Auto-Batch	8.72	75918	114.81

表 5 CUDA 核函数的总执行时间、执行次数和平均执行时间

Table 5　Total execution duration, times and average duration of CUDA kernel functions

方法	总时间/s	执行次数	平均执行时间/ns
No Batch	88.85	9145857	9.72
Auto-Batch	1.26	116883	10.79

图 3 显示当不使用显存池时，显存的分配与释放占训练时间的 56.87%~74.72%，成为性能瓶颈，而显存池显著降低了它的时长。使用我们的显存池时，分配与释放显存的速度是使用 cnmem 时的 5.11~37.11 倍，训练速度是无显存池时的 3.19~3.37 倍，使用 cnmem 时的 1.13~2.63 倍。这表明我们的显存池是高效的。

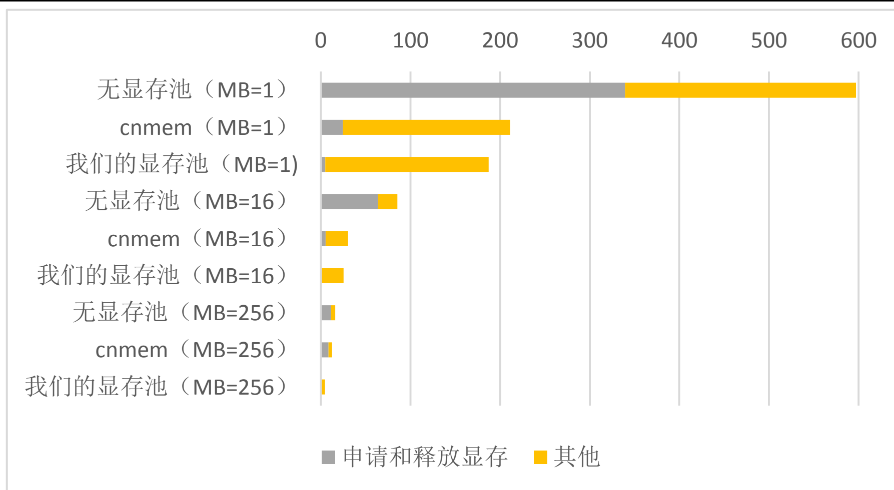


图 3　无显存池、cnmem 和我们的显存池训练时间对比

Fig. 3　Comparison of training time among the absence of the memory pool, cnmem and ours

6　结论

为方便在自然语言处理任务中应用深度学习，移除手动批量化过程，我们提出了轻量级自然语言处理深度学习库 N3LDG。我们仿照 Neubig 等[13]的方法，实现了自动批量化，并且在 CPU 和 GPU 上都高效实现了常见的深度学习计算过程。实验表明，自动批量化显著提高了 CPU 和 GPU 上的执行速度。我们的库在 CNN、Bi-LSTM、Tree-LSTM 模型中的 CPU 性能，以及在 CNN、Tree-LSTM 模型中的 GPU 性能都优于 PyTorch。作为轻量级的库，我们为自然语言处理领域的研究者提供了新的选择。

参考文献

[1] Young T, Hazarika D, Poria S, et al. Recent trends in deep learning based natural language processing [EB/OL]. (2017–08–09) [2018–04–01]. https://arxiv.org/abs/1708.02709

[2] Team T T D, Al-Rfou R, Alain G, et al. Theano: a python framework for fast computation of mathematical expressions [EB/OL]. (2016–03–09) [2018–04–01]. https://arxiv.org/abs/1605.02688

[3] Yu D, Eversole A, Seltzer M, et al. An introduction to computational networks and the computational network toolkit. Microsoft Technical Report MSR-TR-2014–112, 2014

[4] Jia Y, Shelhamer E, Donahue J, et al. Caffe: convolutional architecture for fast feature embedding // Proceedings of the 22nd ACM international conference on Multimedia. Orlando, 2014: 675–678

[5] Abadi M, Barham P, Chen J, et al. Tensorflow: a system for large-scale machine learning // OSDI. Savannah, 2016: 265–283

[6] Paszke A, Gross S, Chintala S, et al. Automatic differentiation in pytorch // NIPS 2017 Autodiff Workshop: The Future of Gradient-based Machine Learning Software and Techniques. Long Beach, 2017: 1–4

[7] Bahdanau D, Cho K, Bengio Y. Neural machine translation by jointly learning to align and translate [EB/OL]. (2014–09–01) [2018–04–01]. https://arxiv.org/abs/1409.0473

[8] Chen Z, Droppo J, Li J, et al. Progressive joint modeling in unsupervised single-channel overlapped speech recognition. IEEE/ACM Transactions on Audio, Speech and Language Processing (TASLP), 2018, 26(1): 184–196.

[9] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition [EB/OL]. (2014–09–04) [2018–04–01]. https://arxiv.org/abs/1409.1556

[10] Chen L C, Papandreou G, Kokkinos I, et al. Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. IEEE transactions on pattern analysis and machine intelligence, 2018, 40(4): 834–848

[11] Looks M, Herreshoff M, Hutchins D L, et al. Deep learning with dynamic computation graphs [EB/OL].

(2017–02–07) [2018–04–01]. https://arxiv.org/abs/1702.02181

[12] Zhang M, Yang J, Teng Z, et al. LibN3L: a lightweight package for neural NLP // LREC. Paris, 2016: 225–229

[13] Neubig G, Goldberg Y, Dyer C. On-the-fly operation batching in dynamic computation graphs [EB/OL]. (2017–05–22) [2018–04–01]. https://arxiv.org/abs/1705.07860

[14] Neubig G, Dyer C, Goldberg Y, et al. Dynet: the dynamic neural network toolkit [EB/OL]. (2017–01–15) [2018–04–01]. https://arxiv.org/abs/1701.03980

[15] Chetlur S, Woolley C, Vandermersch P, et al. Cudnn: efficient primitives for deep learning [EB/OL]. (2014–10–03) [2018–04–01]. https://arxiv.org/abs/1410.0759

[16] Guennebaud G, Jacob B. Eigen [EB/OL]. [2018-04-01]. http://eigen. tuxfamily. org

[17] Knowlton K C. A fast storage allocator. Communications of the ACM, 1965, 8(10): 623–624

[18] Fatica M, LeGresley P, Buck I, et al. High performance computing with CUDA. Tutorial in IEEE Supercomputing , 2007, 18 (6): 397–412

[19] Tai K S, Socher R, Manning C D. Improved semantic representations from tree-structured long short-term memory networks [EB/OL]. (2015–02–28) [2018–04–01]. https://arxiv.org/abs/1503.00075