中文情感分类：实验报告

刘泓尊　2018011446　计 84

liu-hz18@mails.tsinghua.edu.cn

2020 年 5 月 31 日

目录

1 实验内容

使用 PyTorch 实现的中文文本 (Sina-news) 情感分类任务，包含 MLP, CNN, RNN, TextCNN, RCNN 以及 BertCNN 等多种 BaseLine, 达到了比较高的准确率。

2 Environment and Requirements

Environment and Requirements

```
Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz
NVIDIA GeForce GTX 1050 Ti
python==3.7.0
 |-- torch==1.5.0
 |-- torchviz==0.0.1
 |-- scipy==1.4.1
 |-- pandas==0.23.4
 |-- numpy==1.18.2
 |-- jieba==0.39
 |-- six==1.14.0
 |-- scikit_learn==0.23.1
```

```
|-- tensorflow_gpu==1.14.0 # for BERT
|-- tensorflow==2.2.0  # for BERT
|-- termcolor==1.1.0   # for BERT
```

3 运行方式

命令行中输入“python main.py [mlp|cnn|rnn|textcnn|rcnn|bert]”可以训练相应的神经网络. main.py 中针对不同网络设置了对应模型的参数，您可以根据需要修改相关参数.

同时将参数配置 (config.json)、日志 (.log)、模型参数 (.pkl)、模型可视化结构 (.pdf)(使用 Graphivz)、使用 TensorBoard 可视化的 Scalar 曲线 (events.out.tfevents.xxx) 等文件保存在/save/... 文件夹下. 您可以在最上层文件夹下执行命令“tensorboard --logdir save/...”来查看 acc, loss, F1, corr 等 scalar 变化的曲线.

4 预处理语料及词向量

将测试集随机分成一半，分别作为验证集和测试集。

输入数据根据 Word2Vec 的词表做了预处理，将 data 和 label 分开，并用词的 id 替换了汉字，保存为.npy 的向量文件，以节省训练时间。如有需要，您可以使用"python main.py pre" 进行预处理，得到./sina 下对应的预处理文件 test_lable.npy, test_id.npy, train_lable.npy, train_id.npy, valid_lable.npy, valid_id.npy.

./word2vec 文件夹下有 sgns.sogounews.bigram-char 词向量文件。使用./load_data.py 脚本将实验语料中的词筛选出来，保存在/word2vec/matrix.npy。对于词向量文件中没有的词，使用所有词向量的平均值作为结果，标记为 <unk>。训练时直接加载对应的词表作为 Embedding 层。效果比没有预训练的 Embedding 层高出很多，下面的网络模型均使用预训练词向量。

5 实现的模型

实现了 **MLP**, **CNN**, **RNN**, **TextCNN**, **RCNN**，并尝试了 **BertCNN**，共 6 个模型。损失函数使用 CrossEntropyLoss(我还测试过 FocalLoss，但是效果没有明显提升), 优化器使用 AdaGrad。我也测试了 SGD，但是收敛速度太慢了，最终没有使用。参数设为: lr=0.01, weight_decay=0.0001, lr_decay=0.0

具体网络结构见下面的介绍。

5.1 MLP(2 layer): Baseline

使用 2 层全联接网络，加上 Dropout 和 BN，使用 ReLU 激活函数. 此外，我将 CrossEntropyLoss 的权重设置为 [1.0, 1.5, 1.5, 0.8, 1.0, 1.5, 1.5, 2.0], 以抵消类别不均衡带来的影响。结构及参数如下 (详细结构见图20):

句子截断长度: max_length = 256

Emebedding(300) → Linear(300*256, 256) → BatchNorm1d → ReLU → Dropout(0.6) → Linear(256, 64) → BatchNorm1d → ReLU → Dropout(0.6) → Linear(64, 8)

5.2 CNN(1 Conv + MaxPooling) + MLP(1 layer): CNN-Base

朴素的 CNN 作为一种 baseline, 用于和 Text-CNN 对比。我简单地对序列进行时序的 Conv1d，卷积核大小为 3，使用最大池化，所以本网络主要抽取短程信息。使用 Dropout=0.5，使用 ReLU 激活函数. CNN 之后加一个 MLP 映射到 8 个输出.

结构及参数如下 (详细结构见图21): 句子截断长度: `max_length = 256`

`Emebedding(300)` → `Conv1d(300, out_channel=512, kernel_size=3)` → `ReLU` → `Dropout(0.5)` → `MaxPool1d(kernel_size=2, stride=2)` → `Linear(512*128, 512)` → `ReLU` → `Dropout(0.5)` → `Linear(512, 8)`

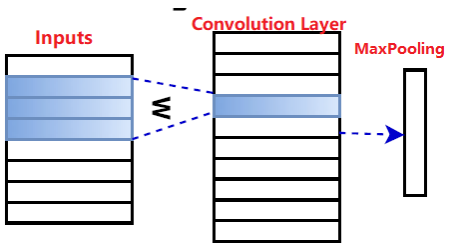

图 1: CNN

5.3 TextCNN: CNN(Multi-Conv + MaxPooling) + MLP(1 layer)

TextCNN 不需要对句子进行截断或 padding，能充分利用句子的信息。我复现了论文 [1, Convolutional Neural Networks for Sentence Classification]，参考了论文 [2, A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification] 的建议进行调参. 最终选择使用长度为 `[2, 3, 4, 5]` 的卷积核各 100 个，提取句子的长程和短程信息. 并在卷积层 (Conv2D) 后加入了 BN 层，最终在句子级别做 `MaxPool1d` 来选择最重要的信息。在输出层依然加了一层 MLP(512,8) 做映射, Dropout=0.5。

网络结构与参数 (详细结构见图23) 为:

`Conv2d(kernel_sizes=[2, 3, 4, 5], out_channel=100)` → `BatchNorm2d` → `ReLU` → `MaxPool1d` → `Dropout` → `Linear(4*100, 512)` → `ReLU` → `Dropout` → `Linear(512, 8)`

模型示意如下:

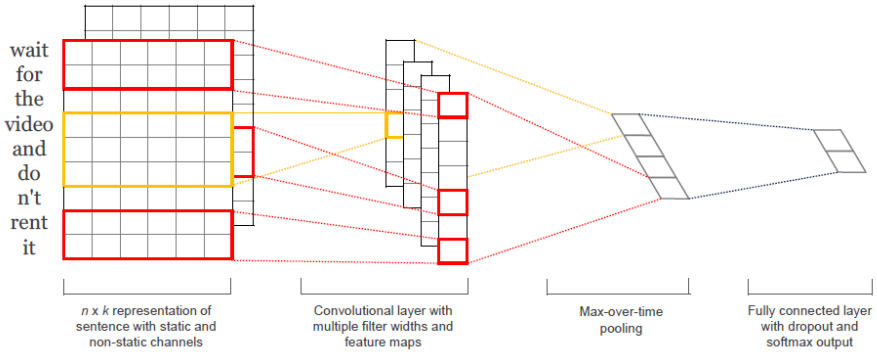

图 2: TextCNN

5.4 bi-(RNN|LSTM|GRU)(2-layer) + MLP(1 layer) + Self-Attention

双向 RNN 网络, 我分别测试了基本 RNN, LSTM, GRU 作为单元的三种网络，并在 RNN 后加了 Self-Attention 机制，参见论文 [3, A Structured Self-attentive Sentence Embedding].

结构及参数 (详细结构见图22) 如下:

`Emebedding(300)` → `bi-RNN(300, hidden_size=256)` → `Self-Attention (da=350, r=30)` → `Linear(30*256*2, 256)` → `ReLU` → `Dropout(0.5)` → `Linear(256, 8)`

网络结构示意如下。当然，也有 Paper 将 Self-Attention 层换成了时序上的 Max-Pooling 层 [4]，同样达到了不错的效果，这里就不详细介绍了。

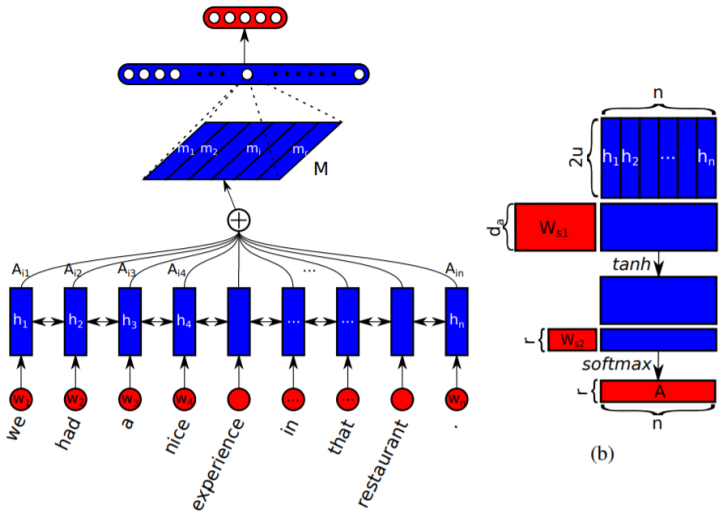

图 3: Self-Attention

5.5 RCNN: bi-(RNN|LSTM|GRU)(2-layer) + MaxPooling + MLP(1 layer)

RCNN 全称 Recurrent Convolutional Neural Networks，我复现了论文 [5, Convolutional Recurrent Neural Networks for Text Classification] 的经典实现, 该网络综合利用了 RNN 和 CNN 的优势,既可以捕捉上下文又兼顾了长程信息。该网络使用先使用 `bi-LSTM(hidden_size=256)` 将输入序列做 Encoding, 之后将 bi-LSTM 的输出与 Embedding 的结果进行连接 `(concat)`, 最后在句子级别做 MaxPool，之后再使用 MLP(512, 8) 映射到输出的 8 个分类, Dropout=0.5.

网络结构 (详细结构见图24) 示意如下:

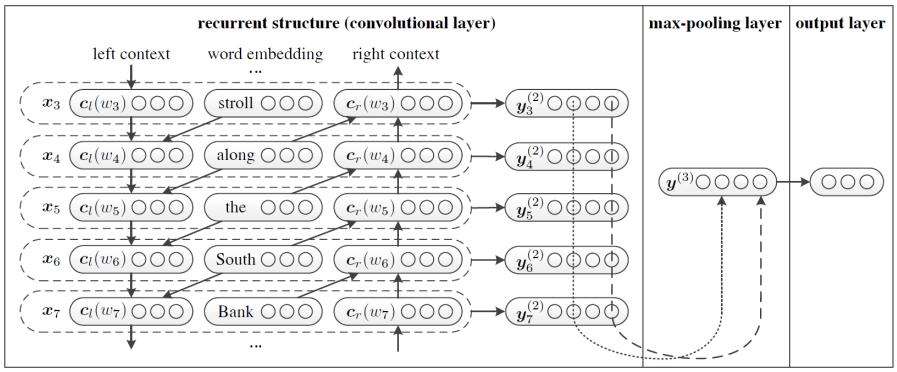

图 4: RCNN

5.6 BERT(Pertrained Sentence Embedding)+ MLP(1 layer)

我尝试了使用预训练模型 BERT 进行训练, 参考了论文 [6, BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding]。不再使用基于 Word2Vec 的 Embedding, 而是使用 BERT 得到**句向量 (dim=768)**，然后将句向量作为输入，经过 `2-layer MLP(768→ 512→ 64→ 8)` 映射之后得到分类结果 (同样加入 `Dropout=0.2, BatchNorm1d`). BERT 使用词嵌入、句嵌入结合的方式得到 Embedding, 可以充分利用全局信息。

通过预处理工作完成从**句子** (进行了截断) 到**句向量**的转化,使用了 Google 提供的 `Chinese Simplified and Traditional, 12-layer, 768-hidden, 12-heads, 110M parameters` 预训练模型chinese_L-12_H-768_A-12[7]. 模型介绍参见Google BERT.

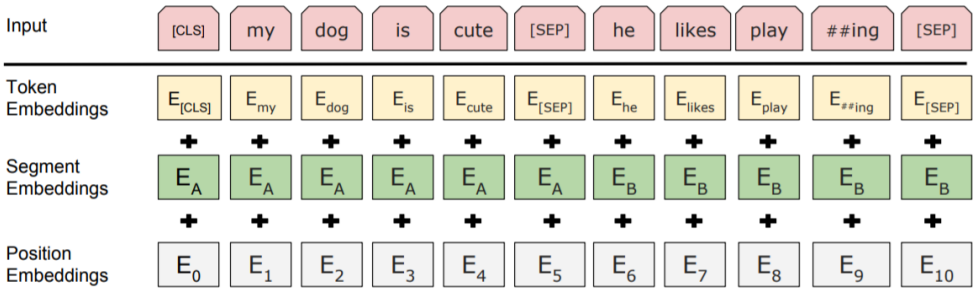

图 5: BERT

6 实验结果及参数对比

下面展示一下不同模型的结果，并选择了一些策略进行对比，做的测试比较多。

6.1 不同模型的效果

6.1.1 Results on Sina-news

Best Performance	test-acc	F1(macro)	CORR	time(s)/epoch	best epoch
MLP	56.64	0.2788	0.3642	1.65	393
BERT-MLP	60.33	0.3205	0.5055	**0.51**	21
CNN	58.86	0.3066	0.4546	2.86	117
Text-CNN	**62.72**	0.3652	**0.5429**	4.03	492
RNN(RNN)	55.09	0.1622	0.2725	3.78	378
RNN(LSTM)	61.16	0.3564	0.4653	12.31	51
RNN(GRU)	**62.64**	**0.3859**	0.4883	4.00	324
RCNN(LSTM)	60.61	0.3342	0.4548	7.72	252
RCNN(GRU)	61.51	0.3149	0.4731	6.64	147

指标均为 Dev-Acc 最大时对测试集 (Test) 的测试结果

效果:

`Text-CNN > RNN(GRU) > RNN(LSTM) > RCNN > BERT-MLP > CNN > MLP > RNN(RNN).`

可以看到 Text-CNN 使用多个卷积核分别捕捉长程、短程信息，比 RNN 效果更好。我认为原因在于文本分类任务对时序依赖不强，更多取决于句子中的关键词，所以 RNN 没有表现

出更好的效果；使用 BERT 预训练的 MLP 的效果比 MLP 好很多, 因为预处理过 Embedding, 所以训练速度最快；

6.1.2 验证集上不同指标的表现 & 测试集准确率

下面一系列图片显示了模型在验证集上的表现，并对比了收敛速度。

可以看到，Text-CNN 效果最好，但是收敛速度很慢，约在 600 个 Epoch 后 Train-Acc 达到 90%. RNN-Cell 产生了梯度消失，但是 LSTM 和 GRU 表现都很好；RCNN 这一组合的模型也在 61% 以上，这为进一步探索模型提供了借鉴；**CNN 系和 RNN 系的准确率都远超 MLP(BaseLine)**, 足见 RNN 和 CNN 对序列信息的捕捉能力很强；在收敛速度上，`MLP > CNN-Base > RNN > RCNN > Text-CNN`。**如果既追求收敛速度又追求准确率，RNN(LSTM 或 GRU) 是个不错的选择。**

三个模型的对比 RNN, CNN, MLP(BaseLine) 三者的效果还是泾渭分明的，这主要是因为 CNN 重点提取局部信息，只有在句子级别做 `MaxPooling` 才能捕捉全局信息；而 RNN 更多采集上下文信息，由其是双向 RNN，对于句间关系捕捉的非常好，但是对于情感分析任务而言，“时序性”并没有那么重要，情感主要体现在关键词中；MLP 尽管可以捕捉全局信息，但是模型过于简单，因此效果不是很好，搭配 BERT 之后效果才勉强和 RNN 相当。更多的讨论可以见我的思考题8.4.

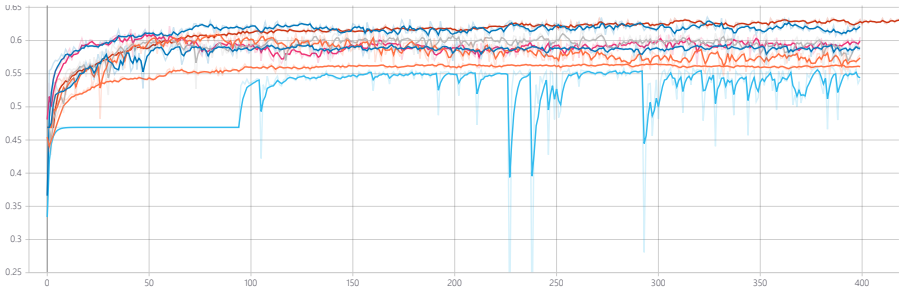

图 6: Dev-Accuracy
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): RNN(GRU), 深蓝色 (下): CNN-Base, 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

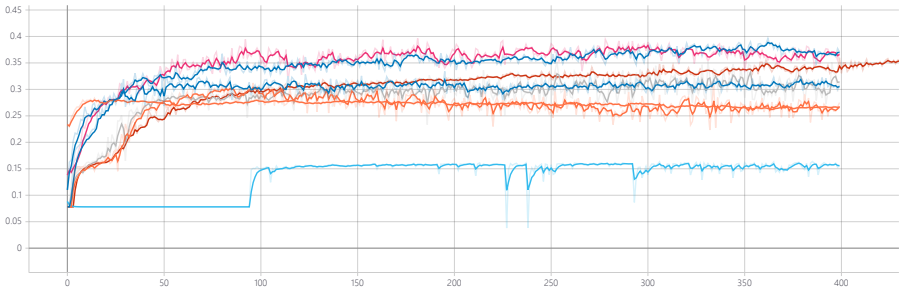

图 7: Dev-F1-Score-Macro
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): RNN(GRU), 深蓝色 (下): CNN-Base, 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

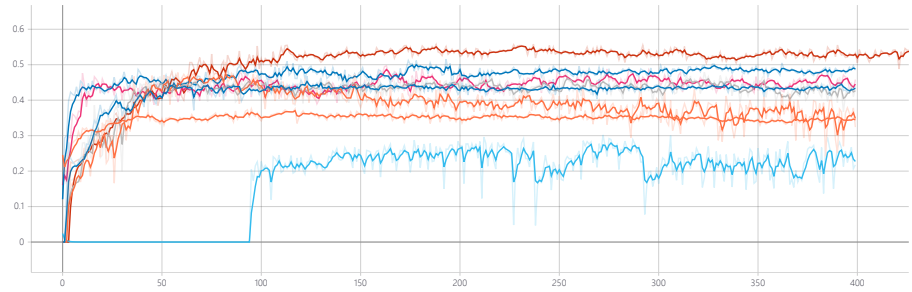

图 8: Dev-Correlation Coefficient
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): RNN(GRU), 深蓝色 (下): CNN-Base, 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

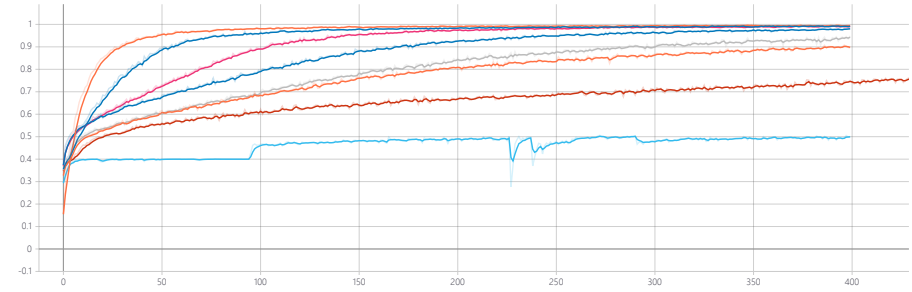

图 9: Train-Accuracy
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): CNN-Base, 深蓝色 (下): RNN(GRU), 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

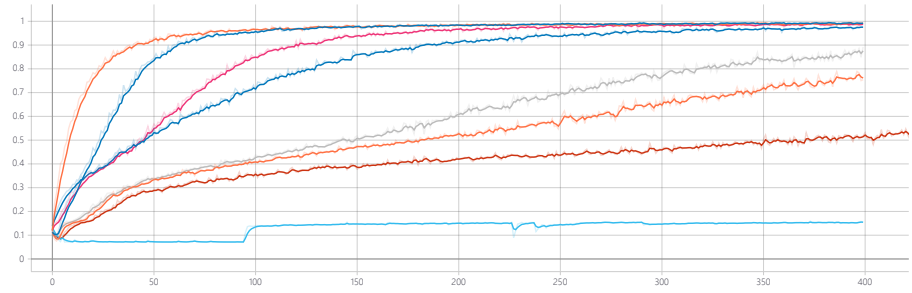

图 10: Train-F1-Score-Macro
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): CNN-Base, 深蓝色 (下): RNN(GRU), 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

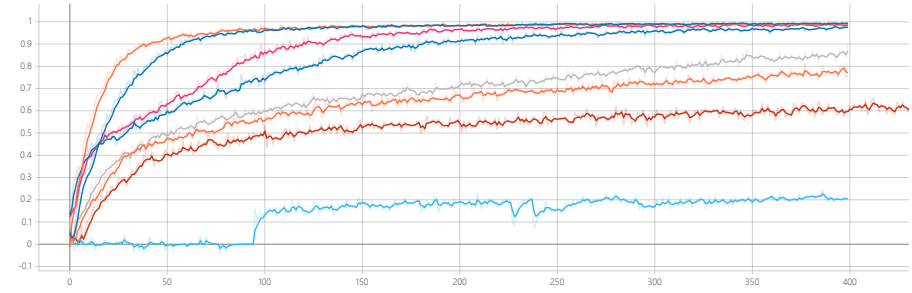

图 11: Train-Correlation Coefficient
橙色 (上):MLP, 橙色 (下):RCNN(GRU), 深蓝色 (上): CNN-Base, 深蓝色 (下): RNN(GRU), 红色: TextCNN, 浅蓝色: RNN(RNN-Cell), 粉色: RNN(LSTM), 灰色:RCNN(LSTM)

6.2 有无 Dropout 和 BN 的对比：以 BERT-MLP 为例

下图展示了有无 Dropout(=0.2) 和 BN 对过拟合程度与收敛速度的影响 (lr=0.01)，以 dev-acc 和 train-acc 为评价指标.

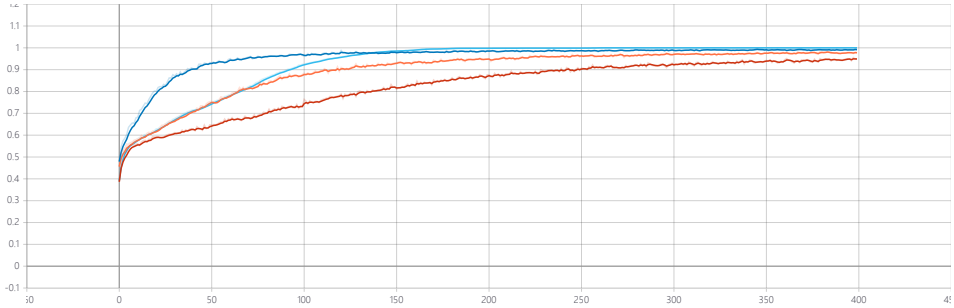

图 12: Train-Acc 曲线 (橙色: only BN，红色: only Dropout, 浅蓝色: no BN or Dropout, 深蓝色: BN and Dropout)

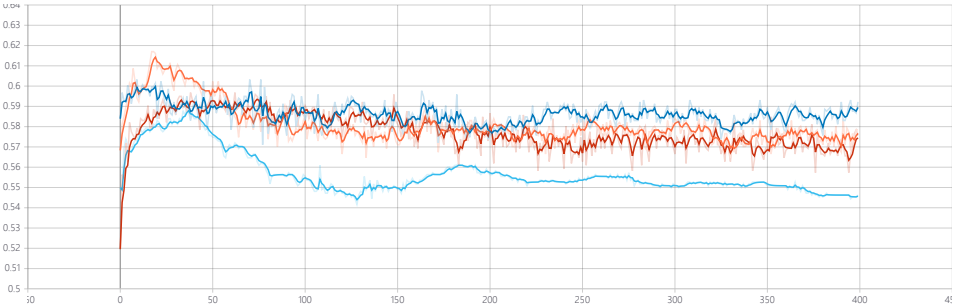

图 13: Dev-Acc 曲线 (橙色: only BN，红色: only Dropout, 浅蓝色: no BN or Dropout, 深蓝色: BN and Dropout)

可以看到 Dropout 和 BN 都可以防止过拟合，都比没有任何 trick 的 MLP 效果好。关于两者的机理我将在思考题中予以解释。本模型中仅使用 BN 效果最好，而 Dropout 和 BN 同时使用可能会适得其反，两者冲突的关键是网络状态切换过程中存在神经元方差 (Neural Variance) 的不一致行为 [8]。我还注意到，尽管 Dropout 和 BN 的单独引入都将使模型收敛速度下降，但是两者同时使用却可以使收敛速度提升，推测是因为 BN 加快收敛的效果胜过了 Dropout.

6.3 句子截断长度的对比：以 MLP/BERT-MLP 为例

对于 MLP 和 CNN(如果不在句子级别做 Pooling) 而言，句子需要进行长度截断或 Padding. 我对比了 MLP 和 BERT-MLP 下句子截断长度对效果的影响。

model & length	64	100	150	200
MLP(256, 64)	55.18	52.14	**56.46**	56.18
BERT-MLP(512, 64)	57.46	58.53	59.22	**60.33**

从表中可以看到，不同的截断长度对效果有明显的影响。因为 BERT 能捕捉句子长程信息，所以句子截断长度越大，效果越好。但是 MLP 对于序列信息不敏感，过长的句子可能会使效果下降。因此，在选择截断长度时，应该考虑到网络对长、短程信息的捕捉能力，选择合适的长度.

6.4 不同 Hidden Size 的对比：以 GRU，MLP 为例

为了探究不同参数规模对结果的影响，我使用 GRU 和 MLP 对不同隐层大小做了测试。结果如下:

Model & Hidden Size	64	128	256	512
MLP	55.52	54.54	**55.64**	54.72
GRU	**63.00**	62.54	62.46	—

隐层大小占据了参数量的绝大部分，更小的参数量有更快的收敛速度。从上表可以看到，隐层规模过小将使得模型的学习能力不强 (欠拟合)，规模过大则使得模型容易陷入过拟合。因此增大隐层数量时，应配合 Dropout 和 BN 层防止过拟合。我在调参过程中还测试了大量不同的隐层规模，在此不一一列出了。

6.5 词向量是否 Fine-tune 的对比：以 MLP, CNN 为例

我采用 Word2Vec 词向量作为 `Embedding`, 对比了 `Embedding` 固定与否对结果的影响，以 dev-acc 为评价指标。结果如下:

Model	Fine-tune	No Fine-tune
MLP	**55.64**	54.54
RNN	61.72	**62.54**

可以看到 MLP 进行 Fine-tune 的效果更好，而 RNN 不进行 Fine-tune 效果更好。我认为此现象的原因是 Word2Vec 预训练的词向量基于大规模语料，得到的词向量质量很高，已经可以代表输入文本的向量表示。而本实验数据集过小，数据分布与 Word2Vec 相差较多，Fine-tune 之后的词向量质量下降了。因此，对于大规模数据集，下游任务可以使用 Fine-tune。但如果数据集比较小，Fine-tune 应慎重。

6.6 是否 Self-Attention 的对比：以 GRU 为例

下图显示了有无 Self-Attention 的 GRU 在 train-acc 和 dev-acc 的表现。

可以看到 Self-Attention 对结果的影响还是比较大的。加入 Self-Attention 后不仅使 Dev-Acc 有了显著上升，防止了过拟合，同时还加快了收敛速度、大幅减少了准确率的波动！我认为 Self-Attention 有很大改进的原因在于：对于情感分类任务，情感往往只存在于句子中某几个词，因此引入 Attention 机制可以让网络专注于提取重点词汇，达到更好的效果.

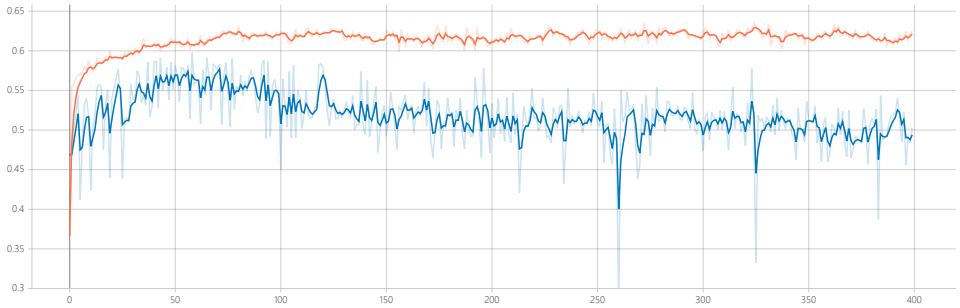

图 14: Dev-Accuracy (橙色: With Attention, 蓝色: Without Attention)

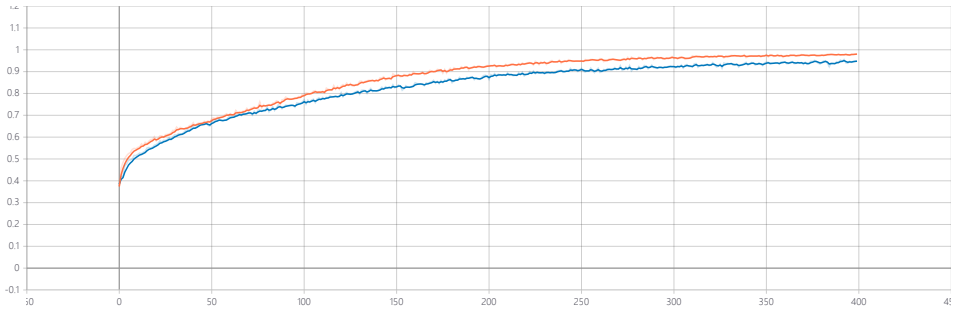

图 15: Train-Accuracy (橙色: With Attention, 蓝色: Without Attention)

6.7 三种 RNN 的对比: RNN, LSTM, GRU

下面对三种 RNN-Cell(`hidden_size=64`) 的指标进行对比:

Model	dev-acc	Time(s)/Epoch
RNN-Cell	55.09	**3.78**
LSTM-Cell	61.16	8.31
GRU-Cell	**63.00**	4.00

从表中可以看到，RNN 的速度最快，GRU 次之，LSTM 最慢，从三种单元的复杂程度便可以得证。RNN 因为只含有一个 Tanh 层，很容易产生梯度消失，在训练后期就“die”了。GRU 和 LSTM 的在本任务上的效果不相上下，GRU 因为参数更小，更容易收敛，但是在大数据集上 LSTM 的能力更强一些。两者都有效防止了梯度消失和梯度爆炸。

6.8 损失函数的比较: 以 CNN-Base 为例

我测试了使用 `CrossEntropyLoss` ,`MSE`, `L1 Loss`, `FocalLoss` 四种损失函数对收敛性和准确率的影响，以 CNN-Base 为例对比 Dev-Acc 和 Train-Acc.

FocalLoss[9] 是为了平衡正负样本不均衡对模型效果带来的影响，而本实验的数据集恰好满足这种特征。但从下图中可以看到 FocalLoss 和 CrossEntropyLoss 并没有很大区别，两者几乎完全一致。`MSE` 的特点是 Loss 值比较大，但是最终的效果并不好，而 `L1` 在一开始就陷入了梯度消失。实际上 MSE 并不适合做分类任务的损失函数，梯度消失现象很严重。

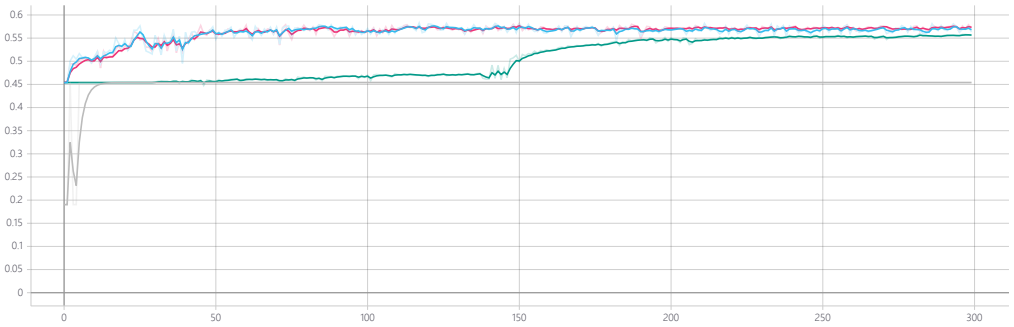

图 16: Dev-Accuracy (蓝色: CrossEntropyLoss, 粉色: FocalLoss, 绿色: MSE, 灰色: L1 Loss)

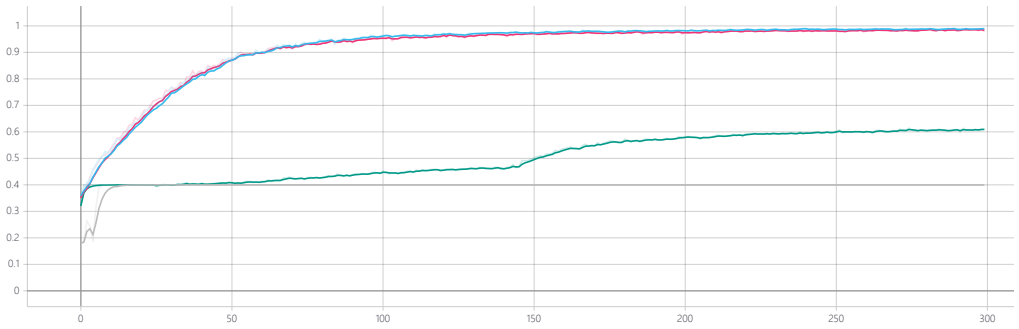

图 17: Trian-Accuracy (蓝色: CrossEntropyLoss, 粉色: FocalLoss, 绿色: MSE, 灰色: L1 Loss)

7 其他尝试

我还做了一些尝试，但是时间有限没有测试很多，所以效果的上限有待验证；有的模型因为数据集限制没有实现。

7.1 上采样 (unsampling) 与下采样 (subsampling)

本次作业的样本很不平衡，“愤怒”一类占了 44%，这对模型效果有很大影响。为了抵消样本不平衡的影响，我对少样本做了上采样 (unsampling)，对“愤怒”一类做了下采样 (subsampling)，使得样本更加均衡。测试发现准确率可以提升 2-3 个点。但是我认为这种修改数据集的方法可能造成同学间的不公平，我没有保留这种方法。

7.2 TF-IDF 加权

TF-IDF 在 NLP 领域用途依然很广，它基于频率统计得到词语在整个段落和语料库中的相对重要性，而且不需要训练过程，可以快速统计，在很多关键词抽取任务上表现不凡。我测试过使用 TF-IDF 作为单词的权重，加权平均 Word2Vec 词向量来得到句向量，但是效果反而下降了。当然，我的方法可能存在问题，仅仅得到句向量等价于 Bag-of-Words 模型，实际上丢失了很多信息，而且对于多分类而言 TF-IDF 可能有局限性，最终没有使用这种方法。

7.3 Multi-Level Attention RNN

其实本次实验我特别想做的就是对输入文本分别做 Character-Level 和 Sentence-Level 的分析，学界对这种结构已经有了很多的实践 [10]。最简单直接的网络是先用一层 RNN 对一个句子 Encoding + Attention, 之后再对段落中若干句子做 Encoding + Attention，网络结构如下：

但是很遗憾数据集中的段落是不含标点的，我使用 `nltk` 做断句得到的效果也并不理想，所以最终没有用上这个 idea。所以我希望明年助教能保留数据集中的标点，给同学们更大的发挥空间！

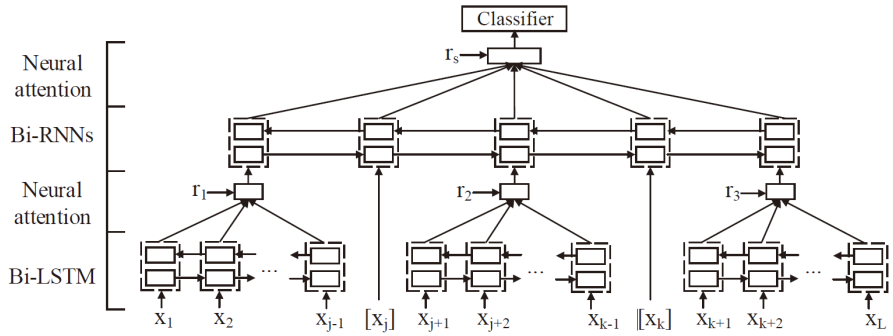

图 18: Multi-Level Attention RNN

8 思考题

8.1 实验训练什么时候停止是最合适的？简要陈述你的实现方式，并试分析固定迭代次数与通过验证集调整等方法的优缺点

结束训练的方式主要有 2 种: (1) 通过若干次测试估计开始过拟合的位置，以此为基准设置固定的 Epoch. (2) 分离出验证集，当验证集准确率最大值比当前准确率高出阈值 △ 时，结束训练；

我采用的是 **(1) 固定迭代次数**的策略，因为本次实验的模型比较小，在我的机器上很快便能跑出结果，这使得我可以测更多的 Epoch, 便于调参与观察。

固定迭代次数 固定迭代次数实现十分简单，而且可以通过准确率的变化来调整参数，使得开发者对模型的掌握更全面，更直观；实际上，固定迭代次数的同时可以将验证集准确率最高的时间点保存下来，作为存档的模型；缺点是对于大模型，训练时间较长。

验证集调整 通过验证集调整可以防止过拟合，同时缩短时间。缺点是不太直观，个人认为适合用在实验后期。此外停止策略需要合理选择，不然很可能使得训练过早停止。

8.2 实验参数的初始化是怎么做的？不同的方法适合哪些地方？

我使用零均值初始化初始化 Linear 层、weight 矩阵和 bias. Self-Attention 部分使用标准正态分布初始化 (randn).

随机初始化十分重要，因为零初始化会导致“对称权重”现象，使得网络无法提取不同的特征。一个比较好的初始化策略是保持每层**输入和输出的方差一致**。

一般来讲，使用零均值均匀或正态分布初始化 Linear 层、weight 矩阵和 bias；根据参数的不同还有很多变种，比如 xavier 的分布适用于激活函数是 Tanh/Sigmoid 的初始化，kaming/He 初始化则更适用于 ReLU；Linear 层一般使用均匀分布初始化；

正交初始化 (Orthogonal) 使用随机初始化的正态分布矩阵进行 QR 分解，使用 Q 矩阵来初始化，适用于 RNN 的初始化，用以解决 RNN 的梯度消失/爆炸问题，也可以去除卷积核的相关性，更快地得到不同特征。

8.3 过拟合是深度学习常见的问题，有什么方法可以方式训练过程陷入过拟合？

扩大数据集是一个比较简单直接的方法，下面主要讨论技术层面的方法。

8.3.1 Dropout

Dropout 在大部分任务上都能有效缓解过拟合。Dropout 不改变网络本身，而是会随机地删除网络中的一般隐藏的神经元，并且让输入层和输出层的神经元保持不变。直观上理解是把参数解耦了。

8.3.2 Batch Normalization(以及其他 Normalization)

Batch Normalization 可以提高网络泛化能力，之后还针对不同网络结构提出了 Layer Normalization，Group Normalization 等归一化方法，都是为了防止过拟合问题。Batch Normalization 将数据拉回到正态分布，减少了层间的 covariate shift, covariate shift 会违背机器学习中的“独立同分布”假设。一定程度上可以防止梯度爆炸和消失问题；在我的测试中，加入 Batch Normalization 还会提高收敛速度。

8.3.3 加入验证集，提前停止 (hold out)

这种方式可以方便我们发现过拟合；在每一个 Epoch 结束之后都跑一遍验证集的准确率，一旦验证集上的准确率出现明显的下降趋势，我们就可以停止训练，这样基本上可以保证测试集准确率较高，但其实并没有改变网络的任何结构。

8.3.4 Regularization

规范化后的神经网络往往有更强的泛化能力。最常见的手段是对矩阵 w 做 weight decay(权重衰减)。比如 L2 Regularization 就是在 Loss 上加一个 $\frac{\lambda}{2n}\sum_w w^2$. 这种策略使得神经网络会在“小权重”与“最小化 Loss”之间找一个折中。

8.4 试分析 CNN, RNN, MLP 三者的优缺点

因为我只进行过 NLP 方面的实验，所以我主要从这个角度做一些解答。

8.4.1 MLP

优点 结构简单，易于理解，可以从函数的角度解释。三层左右带非线性激活函数的 MLP 基本就可以拟合绝大部分函数；基本上是矩阵运算，可以实现较大的加速，可并行；同样的任务下，模型参数量相对较小，训练速度快；

缺点 输入大小固定，处理句子等序列问题需要长度截断，可能会丢失一些信息；提取信息的能力较弱，容易发生过拟合 (容易陷入局部最优解)；对于 NLP 任务，一般是用 Bag-of-Word-Vectors 简单将词向量相加，然后使用 MLP，无法利用上下文信息；

8.4.2 CNN

优点 很方便提取局部信息，可解释，卷积核还和 n-gram 模型形成了对应；Conv2d 等卷积操作可以转化为矩阵乘法，能矩阵加速，可并行；以权值共享的方式缓解了参数膨胀问题；

缺点 对于一般的 CNN，输入大小依然是固定的，所以对序列问题可能需要长度截断 (但是 TextCNN 通过对整个句子做 Pooling 解决了这个问题，也给 CNN 带来了更多的灵活性，详见我的实现)；注重局部信息 (类似于 N-gram)，卷积核过小则不能把握长距离信息，对于长句子级别的“转折”等关系无法做出反应；

8.4.3 RNN

优点 很自然地和序列问题形成映射，可以利用上下文信息 (比如 bi-RNN 就可以利用过去和将来的信息)；输入长度可变，不需要做句子截断；擅长把句子做 decoding.

缺点 在时间轴上容易产生梯度消失/梯度爆炸；简单的 RNN 无法解决长程依赖问题 (LSTM 解决了此问题)；在时间序列上无法并行，训练速度较慢。

关于三种 RNN Cell 的对比我在6.2部分已经阐述，在此不再重复。

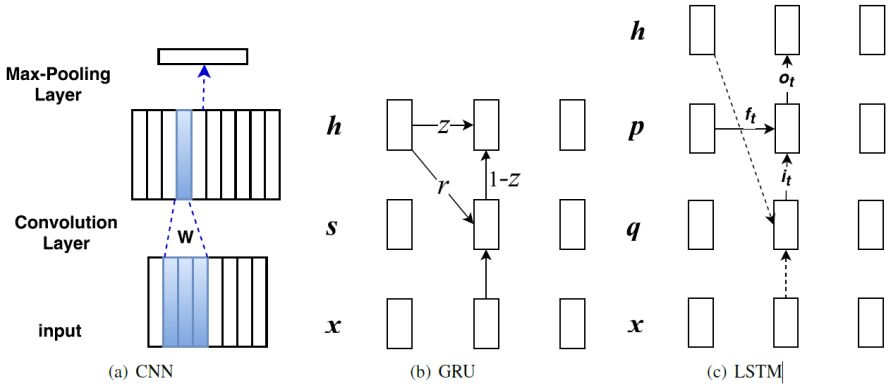

图 19: CNN 和 RNN[11]

9 实验总结

本次实验我完整的实现了 MLP, CNN(Text-CNN), RNN(LSTM, GRU) 的模型，对我这个从没有接触过深度学习框架的小白来说还是有很大的挑战，但是好在 `PyTorch` 有详细的文档，API 也比较清晰，所以把原理弄懂之后很快便搭好了网络。同时我还实现了 Text-CNN 和 RCNN，并使用 Google 发布的 BERT 测试了一个简单的 MLP，领略了预训练以及模型组合的效果。我的收获真的很大：除了熟悉了 `PyTorch` 框架、实战了众多经典网络之外，我还阅读了大量的文献，锻炼了我的文献阅读能力。

因为初涉深度学习，所以在调参上没有太多经验，只能按照一些论文里的建议去调参，而且 6 个模型给我带来的调参任务比较重，可能并没有调到最好，但是我确实感受到了 MLP, CNN, RNN 之间的差异性。但是作为一个初学者，我认为从宏观上把握不同网络的特征，体会不同 trick 的机理是更重要的，毕竟网络结构决定了模型的上限，而调参只能有几个点的提升。

此外，我感到训练集和测试集的数据分布不太一样，最终 dev acc 最好也只能达到 61-63%。而且我统计发现样本类别及其不均衡，“愤怒”类别占了将近一半。在不改变数据集的情况下，我尝试使用 Focal Loss[9] 来抵消类别不均衡的影响，但准确率没有明显提升；而做上采样和下采样又引入了不公平因素，所以希望助教老师能传授一些有效的方法，谢谢！

感谢马老师和助教的悉心指导！

参考文献

[1] Y. Kim, "Convolutional neural networks for sentence classification," pp. 1746–1751, oct 2014.

[2] Y. Zhang and B. Wallace, "A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification," 2015.

[3] Z. Lin, M. Feng, C. N. dos Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio, "A structured self-attentive sentence embedding," 2017.

[4] D. Zhang and D. Wang, "Relation classification via recurrent neural network," 2015.

[5] R. Wang, Z. Li, J. Cao, T. Chen, and L. Wang, "Convolutional recurrent neural networks for text classification," in *2019 International Joint Conference on Neural Networks (IJCNN)*, 2019.

[6] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2018.

[7] Y. Cui, W. Che, T. Liu, B. Qin, Z. Yang, S. Wang, and G. Hu, "Pre-training with whole word masking for chinese bert," 2019.

[8] X. Li, S. Chen, X. Hu, and J. Yang, "Understanding the disharmony between dropout and batch normalization by variance shift," 2018.

[9] T.-Y. Lin, P. Goyal, R. Girshick, K. He, and P. Dollár, "Focal loss for dense object detection," 2017.

[10] M. Xiao and C. Liu, "Semantic relation classification via hierarchical recurrent neural network with attention," in *Proceedings of COLING 2016, the 26th International Conference on Computational Linguistics: Technical Papers*, (Osaka, Japan), pp. 1254–1263, The COLING 2016 Organizing Committee, 2016.

[11] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 2015.

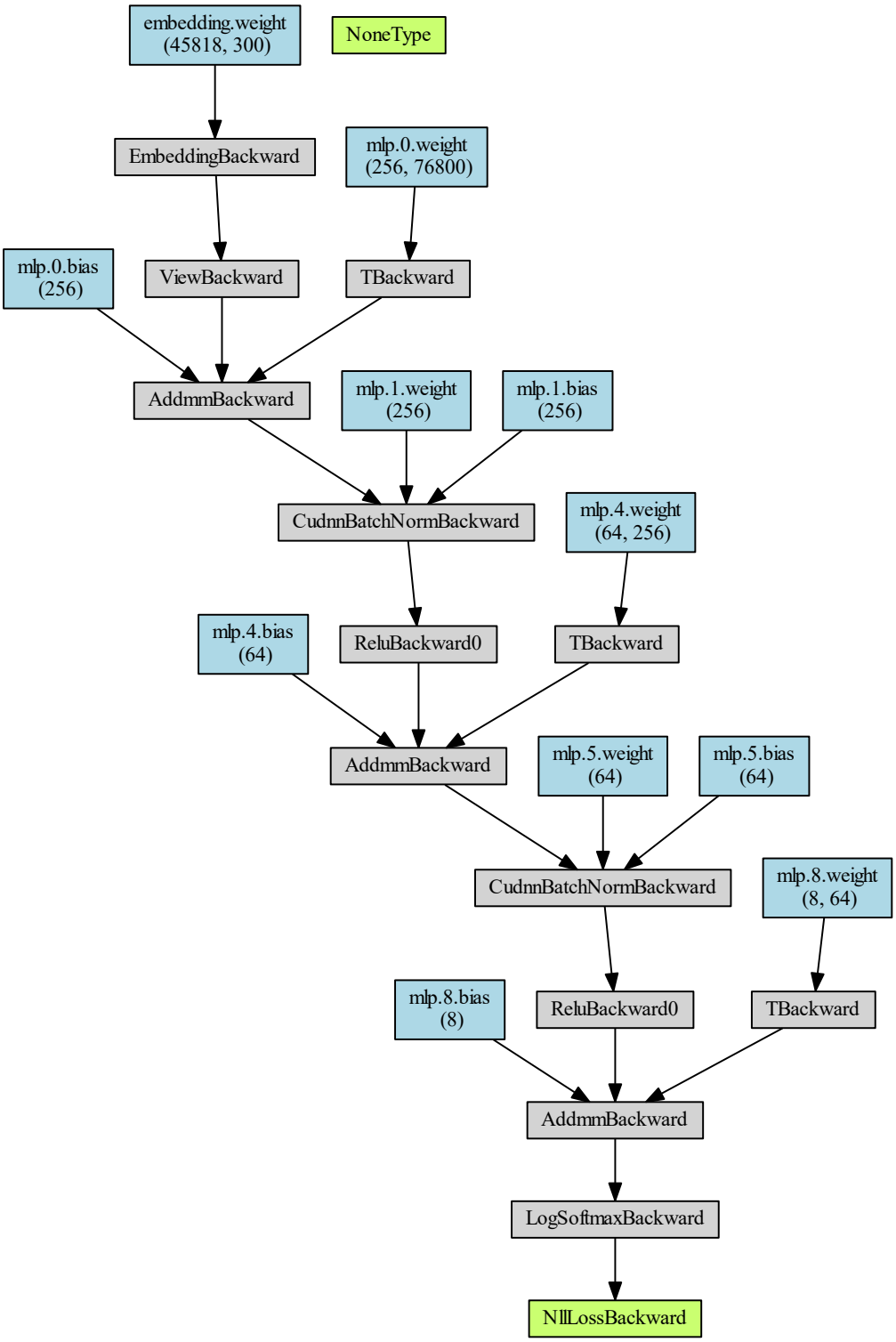
embedding.weight
(45818, 300)
NoneType
EmbeddingBackward
mlp.0.weight
(256, 76800)
mlp.0.bias
(256)
ViewBackward
TBackward
AddmmBackward
mlp.1.weight
(256)
mlp.1.bias
(256)
CudnnBatchNormBackward
mlp.4.weight
(64, 256)
mlp.4.bias
(64)
ReluBackward0
TBackward
AddmmBackward
mlp.5.weight
(64)
mlp.5.bias
(64)
CudnnBatchNormBackward
mlp.8.weight
(8, 64)
mlp.8.bias
(8)
ReluBackward0
TBackward
AddmmBackward
LogSoftmaxBackward
NllLossBackward

图 20: MLP

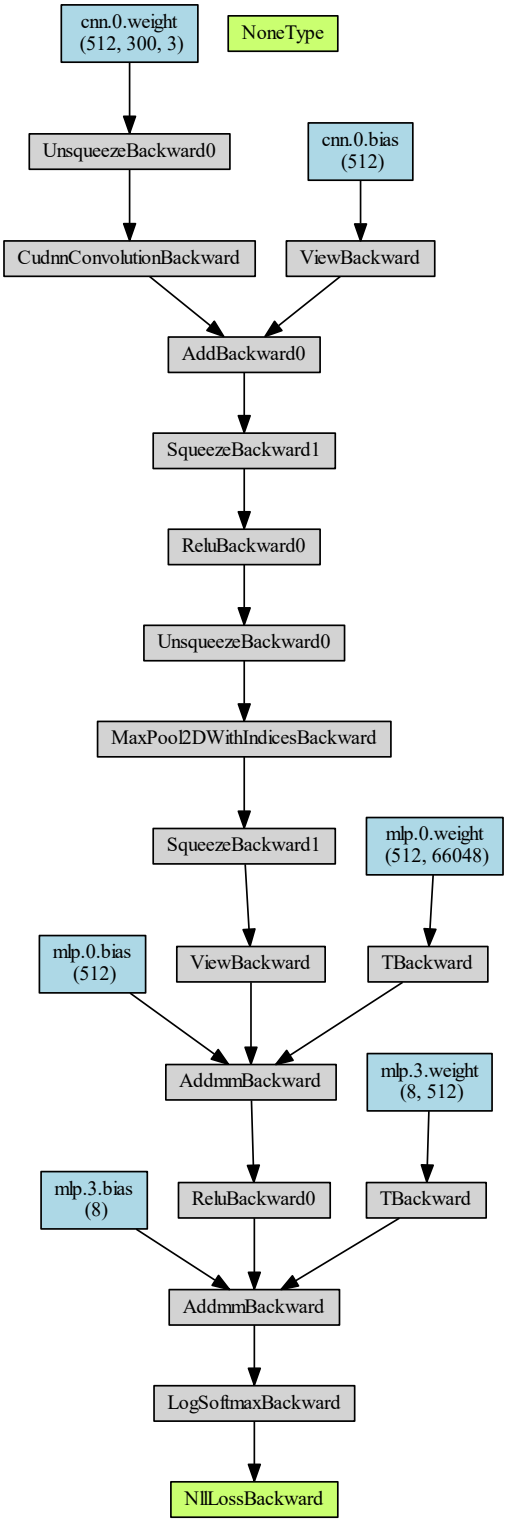
cnn.0.weight
(512, 300, 3)
NoneType
UnsqueezeBackward0
cnn.0.bias
(512)
CudnnConvolutionBackward
ViewBackward
AddBackward0
SqueezeBackward1
ReluBackward0
UnsqueezeBackward0
MaxPool2DWithIndicesBackward
SqueezeBackward1
mlp.0.weight
(512, 66048)
mlp.0.bias
(512)
ViewBackward
TBackward
AddmmBackward
mlp.3.weight
(8, 512)
mlp.3.bias
(8)
ReluBackward0
TBackward
AddmmBackward
LogSoftmaxBackward
NllLossBackward

图 21: CNN-Base

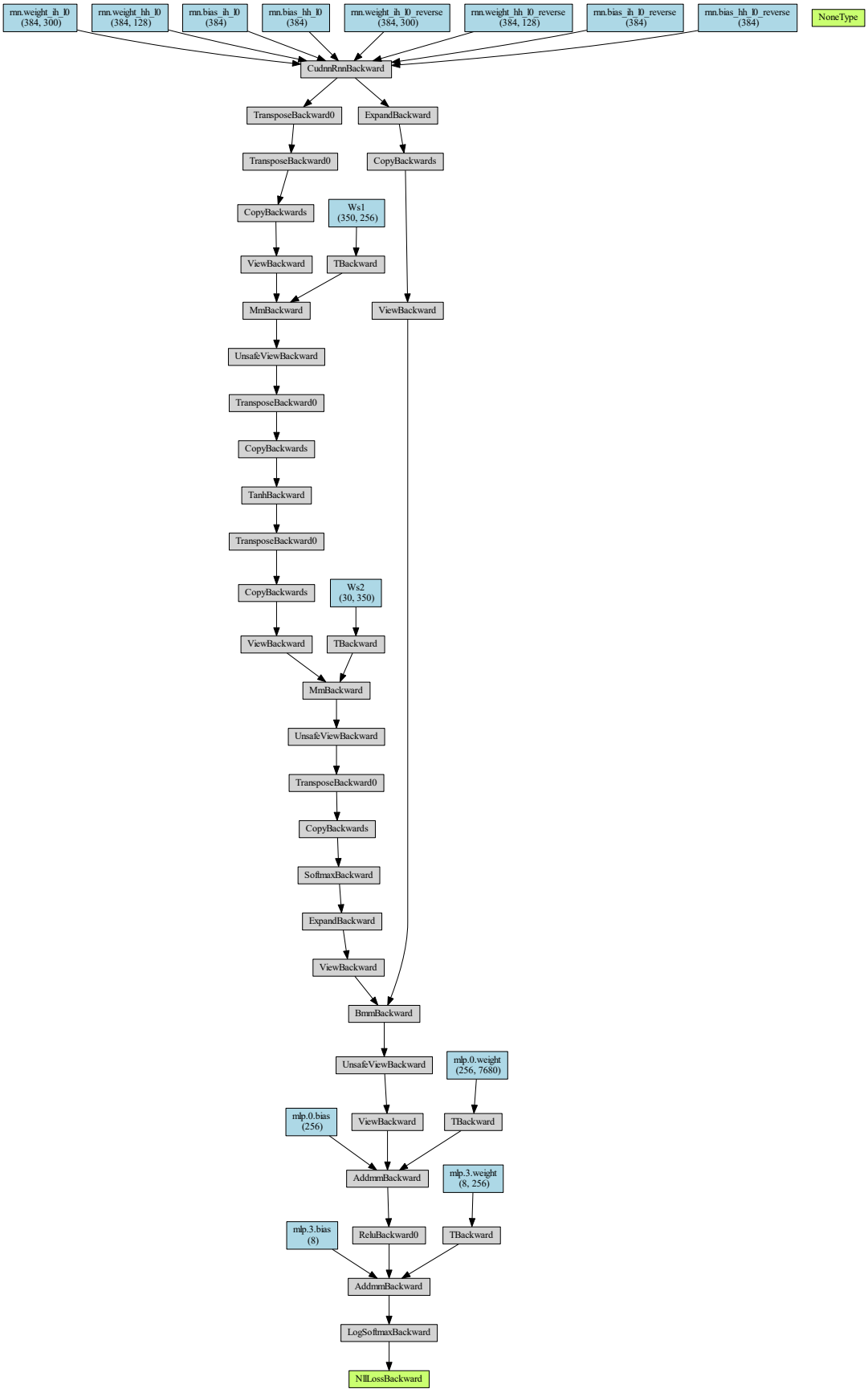
rnn.weight_ih_l0 (384, 300)
rnn.weight_hh_l0 (384, 128)
rnn.bias_ih_l0 (384)
rnn.bias_hh_l0 (384)
rnn.weight_ih_l0_reverse (384, 300)
rnn.weight_hh_l0_reverse (384, 128)
rnn.bias_ih_l0_reverse (384)
rnn.bias_hh_l0_reverse (384)
NoneType
CudnnRnnBackward
TransposeBackward0
ExpandBackward
TransposeBackward0
CopyBackwards
CopyBackwards
Ws1 (350, 256)
ViewBackward
TBackward
MmBackward
ViewBackward
UnsafeViewBackward
TransposeBackward0
CopyBackwards
TanhBackward
TransposeBackward0
CopyBackwards
Ws2 (30, 350)
ViewBackward
TBackward
MmBackward
UnsafeViewBackward
TransposeBackward0
CopyBackwards
SoftmaxBackward
ExpandBackward
ViewBackward
BmmBackward
UnsafeViewBackward
mlp.0.weight (256, 7680)
mlp.0.bias (256)
ViewBackward
TBackward
AddmmBackward
mlp.3.weight (8, 256)
mlp.3.bias (8)
ReluBackward0
TBackward
AddmmBackward
LogSoftmaxBackward
NllLossBackward

图 22: bi-RNN

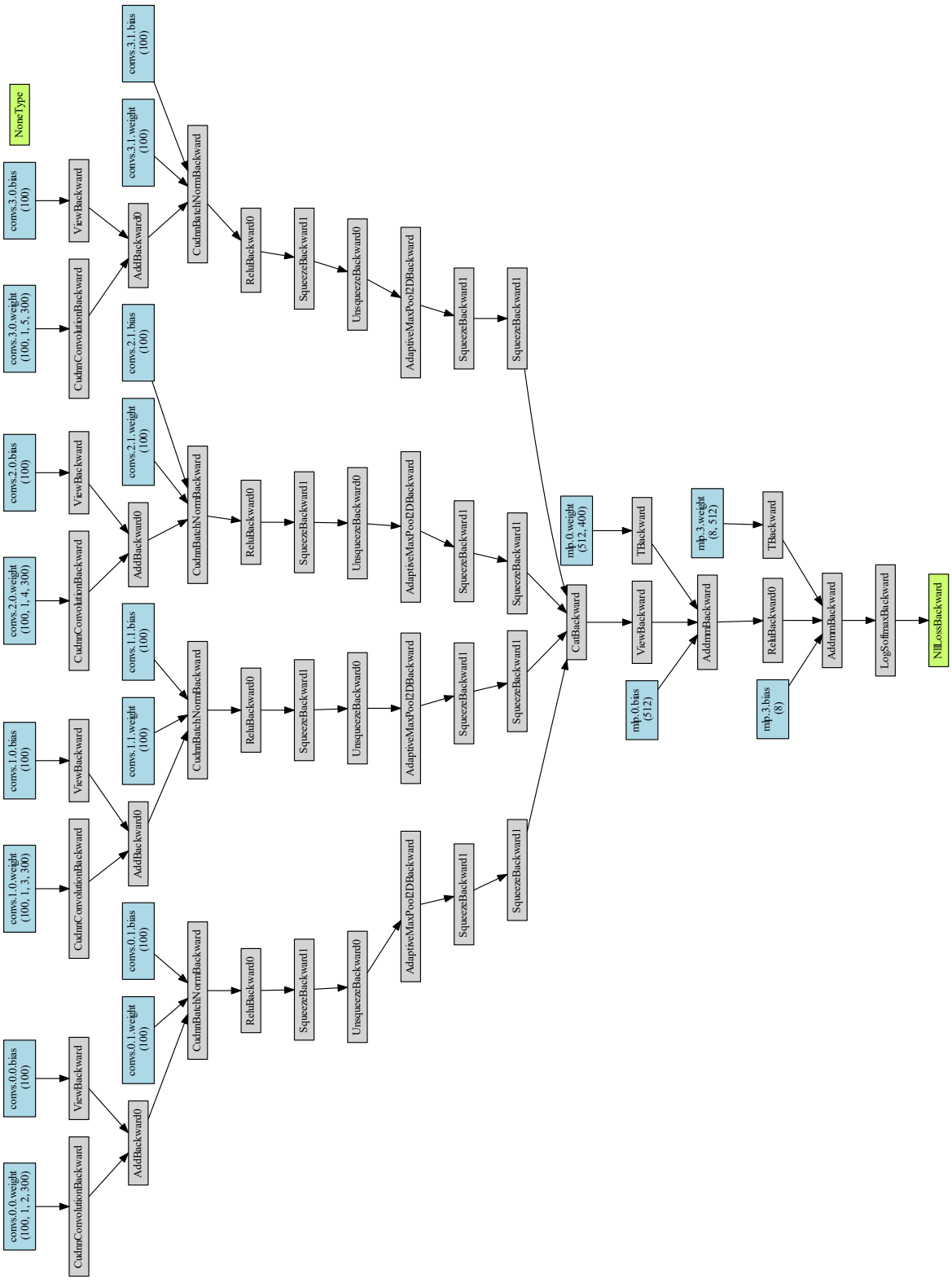
convs.0.weight
(100, 1, 2, 300)
CudnnConvolutionBackward
convs.0.bias
(100)
ViewBackward
AddBackward0
convs.0.1.weight
(100)
convs.0.1.bias
(100)
CudnnBatchNormBackward
ReluBackward0
SqueezeBackward1
UnsqueezeBackward0
AdaptiveMaxPool2DBackward
SqueezeBackward1
SqueezeBackward1
convs.1.0.weight
(100, 1, 3, 300)
CudnnConvolutionBackward
convs.1.0.bias
(100)
ViewBackward
AddBackward0
convs.1.1.weight
(100)
convs.1.1.bias
(100)
CudnnBatchNormBackward
ReluBackward0
SqueezeBackward1
UnsqueezeBackward0
AdaptiveMaxPool2DBackward
SqueezeBackward1
SqueezeBackward1
convs.2.0.weight
(100, 1, 4, 300)
CudnnConvolutionBackward
convs.2.0.bias
(100)
ViewBackward
AddBackward0
convs.2.1.weight
(100)
convs.2.1.bias
(100)
CudnnBatchNormBackward
ReluBackward0
SqueezeBackward1
UnsqueezeBackward0
AdaptiveMaxPool2DBackward
SqueezeBackward1
SqueezeBackward1
convs.3.0.weight
(100, 1, 5, 300)
CudnnConvolutionBackward
convs.3.0.bias
(100)
ViewBackward
AddBackward0
convs.3.1.weight
(100)
convs.3.1.bias
(100)
CudnnBatchNormBackward
ReluBackward0
SqueezeBackward1
UnsqueezeBackward0
AdaptiveMaxPool2DBackward
SqueezeBackward1
SqueezeBackward1
NoneType
CatBackward
mlp.0.weight
(512, 400)
TBackward
ViewBackward
mlp.0.bias
(512)
AddmmBackward
mlp.3.weight
(8, 512)
TBackward
ReluBackward0
mlp.3.bias
(8)
AddmmBackward
LogSoftmaxBackward
NllLossBackward

图 23: TextCNN

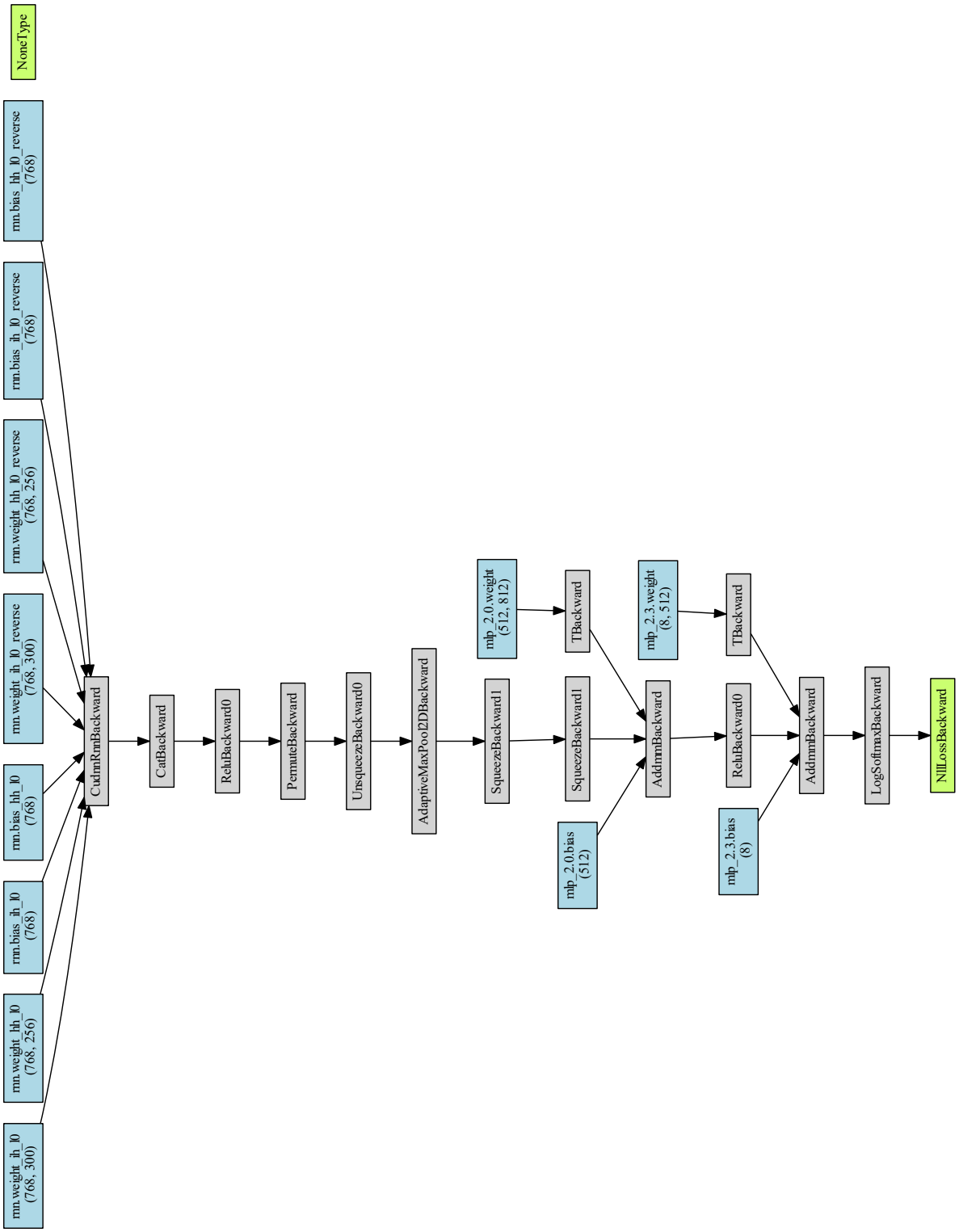
NoneType
rnn.bias_hh_l0_reverse (768)
rnn.bias_ih_l0_reverse (768)
rnn.weight_hh_l0_reverse (768, 256)
rnn.weight_ih_l0_reverse (768, 300)
rnn.bias_hh_l0 (768)
rnn.bias_ih_l0 (768)
rnn.weight_hh_l0 (768, 256)
rnn.weight_ih_l0 (768, 300)
CudnnRnnBackward
CatBackward
ReluBackward0
PermuteBackward
UnsqueezeBackward0
AdaptiveMaxPool2DBackward
SqueezeBackward1
SqueezeBackward1
mlp_2.0.bias (512)
mlp_2.0.weight (512, 812)
TBackward
AddmmBackward
mlp_2.3.weight (8, 512)
TBackward
ReluBackward0
mlp_2.3.bias (8)
AddmmBackward
LogSoftmaxBackward
NllLossBackward

图 24: RCNN