不同词向量编码与解码器结构在IMDb数据集上的表现分析

报告人：陈睿祺

日期：2020年4月5日

北京大学

内容提要

- 背景
- 模型和方法
- 实验分析与展示

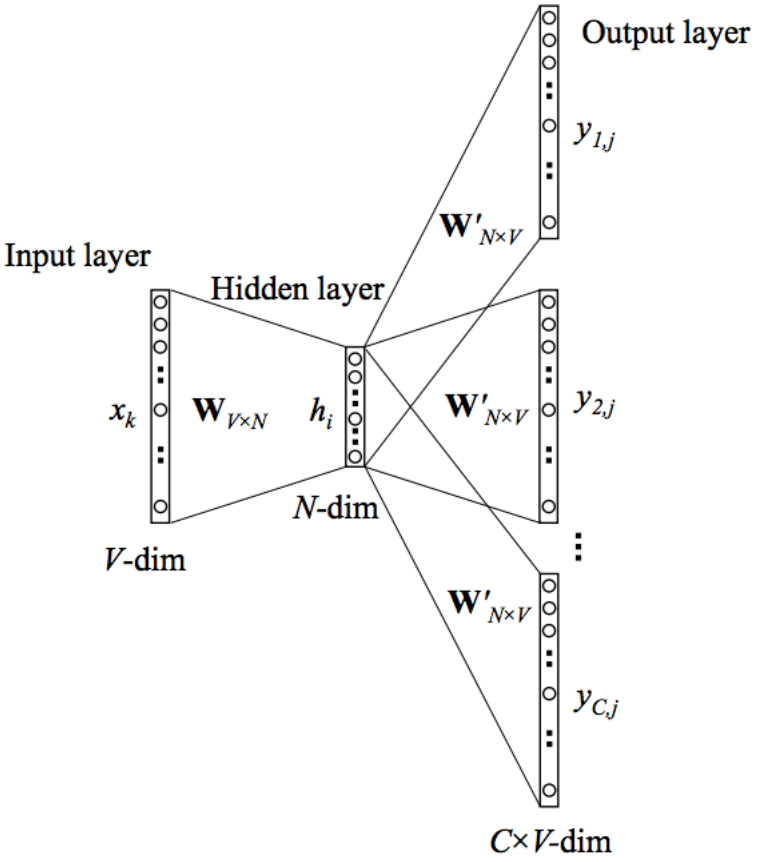
背景

- IMDb数据集
  - 文本情感判断任务
  - 训练集和测试集各25000条影评
- 循环神经网络
  - 每一时刻的输入都包括之前步骤的结果
  - 最适合时序信息（如文本）处理
- 词向量编码
  - 将文本表示为方便处理的数字向量
  - 希望表示出词汇之间的语义联系

模型和方法

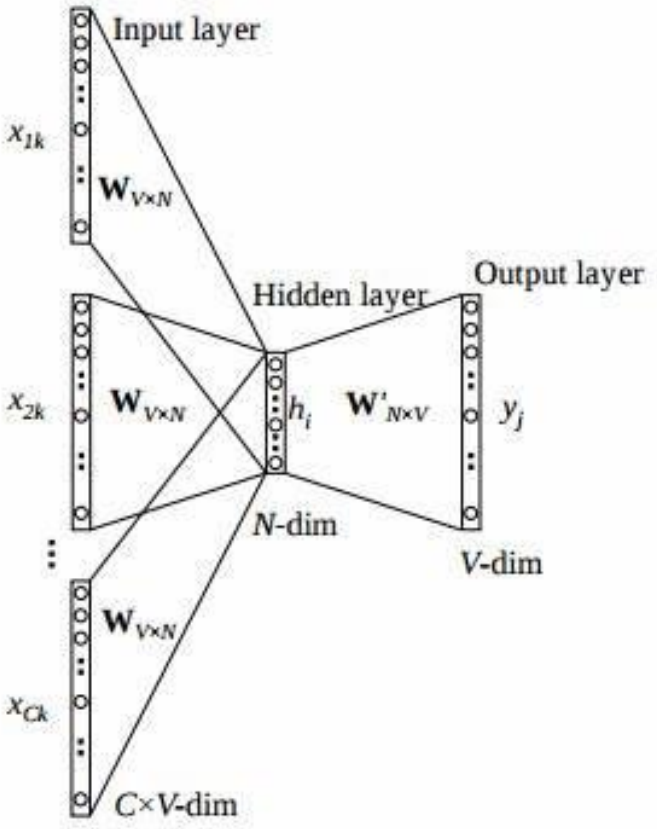
- 词向量编码
  - Skip-gram
  - CBOW
  - Task-oriented
- 解码器结构
  - LSTM
  - GRU
  - Dual-layer RNN

Skip-gram



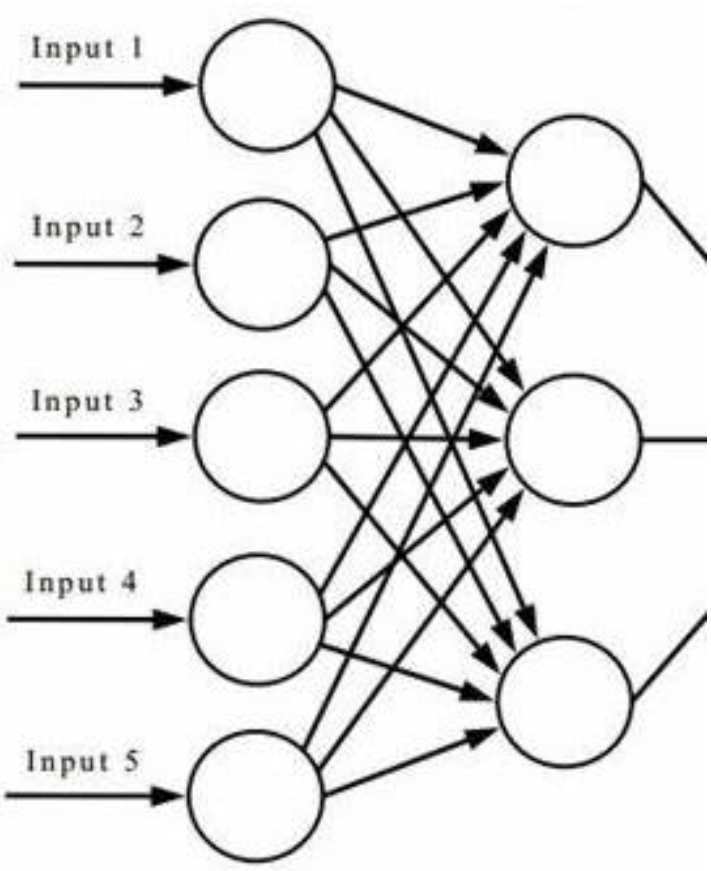
- 输入为One-hot编码，输出可以理解为概率分布
- 隐层不设激活函数，输出层为Softmax
- 用当前单词预测其周围的C个单词，最后提取隐层表示作为词向量

CBOW



- 与Skip-gram恰好相反，使用周围的单词编码当前单词

Task-Oriented



- 简单的全连接层
- 直接与解码器一起训练生成表示

编码器训练

- 三种编码方法都采用5000维One-hot编码输入和64维词向量输出
- 使用Keras的Embedding层编码
  - Skip-gram和CBOW使用Gensim实现，将权重导入Keras并设置为不可训练
  - Task-oriented直接使用Embedding层训练
- 使用相同大小的LSTM+Softmax分类器进行调参和性能评价

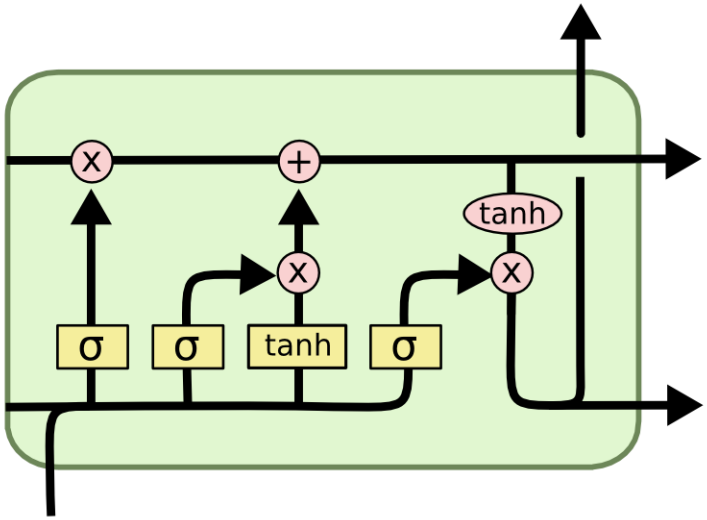
Skip-gram

```
# ---------------- Skip-gram --------------- #


print("Training...")
if not os.path.isfile(datPath + sgFile):
    skGram = []
    for curr, e in zip(sgLoss, sgEpoch):
        skGram.append(word2vec.Word2Vec(wvTrain, size=nDim,
            min_count=1, sg=1, callbacks=[callback(curr)],
            iter=e, compute_loss=True))
    if not os.path.isdir(datPath):
        os.mkdir(datPath)
    with open(datPath + sgFile, 'wb') as f:
        dump((sgLoss, sgEpoch, skGram), f)
else:
    with open(datPath + sgFile, 'rb') as f:
        (sgLoss, sgEpoch, skGram) = load(f)
```

CBOW

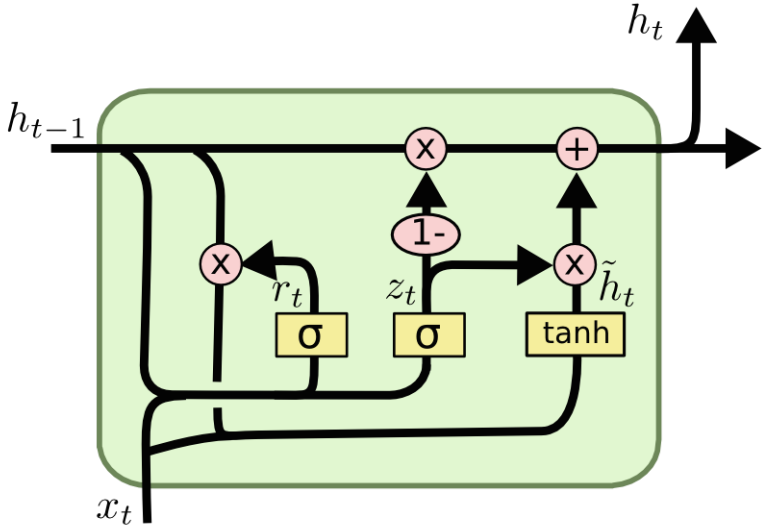
```
# ---------------- CBOW --------------- #


print("Training...")
if not os.path.isfile(datPath + cbowFile):
    cbowGram = []
    for curr, e in zip(cbowLoss, cbowEpoch):
        cbowGram.append(word2vec.Word2Vec(wvTrain, size=nDim,
            min_count=1, sg=0, callbacks=[callback(curr)],
            iter=e, compute_loss=True))
    if not os.path.isdir(datPath):
        os.mkdir(datPath)
    with open(datPath + cbowFile, 'wb') as f:
        dump((cbowLoss, cbowEpoch, cbowGram), f)
else:
    with open(datPath + cbowFile, 'rb') as f:
        (cbowLoss, cbowEpoch, cbowGram) = load(f)
```

LSTM



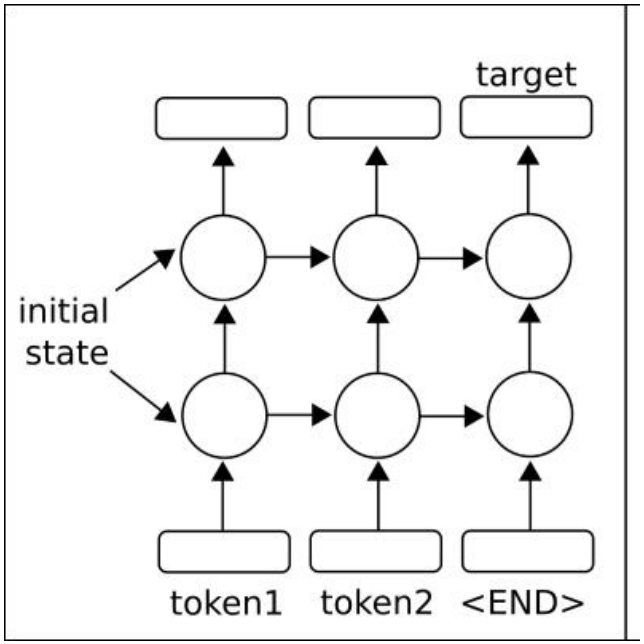
- 下方的箭头为“短时记忆”，上方的箭头为“长时记忆”
- “长时记忆”只进行少量线性操作，故可以长期保存信息，又可以通过“遗忘门”“更新门”更新
- “长时记忆”通过输出门影响“短时记忆”

GRU



- 合并了长时与短时记忆
- 合并了遗忘门与更新门
- 参数比LSTM更少

Dual-layer RNN



- 两层简单的RNN，上一时刻输出通过tanh加入当前时刻的输入
- 第一层RNN输出一个序列，第二层只输出更新至最后一个token的结果

解码器训练

- 都使用5000-to-64的Embedding层
  - 比通行的模型要小很多，但表现仍然不错
- 参数量都控制在75k左右
  - LSTM: 64-to-108
  - GRU: 64-to-128
  - RNN: 64-to-200-to-80
- 都使用Softmax分类器和交叉熵Loss
- 都使用Adam优化并对初始学习率、Batch size、Dropout ratio进行调参

LSTM

```
# ---------------- LSTM ------------------- #

print('Building LSTM ...')
lstmM = Sequential()
lstmM.add(Embedding(nWord, nDim))
lstmM.add(LSTM(lstmOut, dropout=lstmDrop, recurrent_dropout=lstmDrop))
lstmM.add(Dense(1, activation='sigmoid'))
lstmM.summary()

lstmOpt = Adam(learning_rate=lstmLR)
lstmM.compile(loss='binary_crossentropy',
    optimizer=lstmOpt, metrics=['accuracy'])

print('Training LSTM...')
lstmHist = lstmM.fit(x_train, y_train,
    batch_size=lstmBatSize, epochs=nEpoch,
    validation_data=(x_test, y_test), callbacks=[reduce_lr])
```

GRU

```
# ------------------ GRU ------------------ #

print('Building GRU ...')
gruM = Sequential()
gruM.add(Embedding(nWord, nDim))
gruM.add(GRU(gruOut, dropout=Drop, recurrent_dropout=Drop))
gruM.add(Dense(1, activation='sigmoid'))
gruM.summary()

gruOpt = Adam(learning_rate=LR)
gruM.compile(loss='binary_crossentropy',
    optimizer=gruOpt, metrics=['accuracy'])

print('Training GRU...')
gruHist = gruM.fit(x_train, y_train,
    batch_size=BatSize, epochs=nEpoch,
    validation_data=(x_test, y_test), callbacks=[reduce_lr])
```

RNN

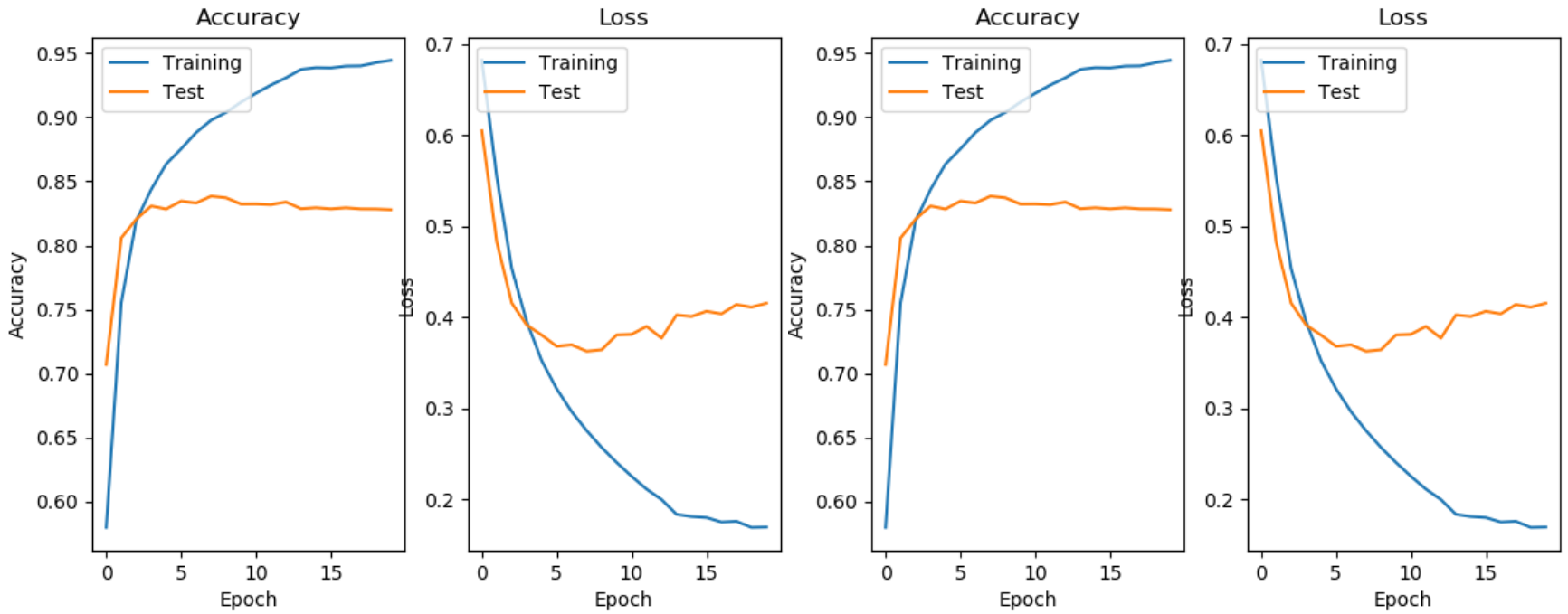
```
# ---------------- RNN --------------------- #

print("Building RNN ...")
rnnM = Sequential();

rnnM.add(Embedding(nWord, nDim))
rnnM.add(SimpleRNN(rnnHid, dropout=Drop, recurrent_dropout=Drop,
return_sequences=True))
rnnM.add(SimpleRNN(rnnOut, dropout=Drop, recurrent_dropout=Drop))
rnnM.add(Dense(1, activation='sigmoid'))
rnnM.summary()

rnnOpt = Adam(learning_rate=LR)
rnnM.compile(loss='binary_crossentropy',
    optimizer=rnnOpt, metrics=['accuracy'])

print('Training RNN ...')
rnnHist = rnnM.fit(x_train, y_train,
    batch_size=BatSize, epochs=nEpoch,
    validation_data=(x_test, y_test), callbacks=[reduce_lr])
```

结果

- 网络大小的选择
- 解码器参数选择
- 解码器性能比较
- 编码器参数选择
- 编码器性能比较
- 讨论

网络大小选择

- 基于Keras的示例（两种参数设置）
  - 20000-to-128 Embedding
  - 128-to-128 LSTM



网络大小选择

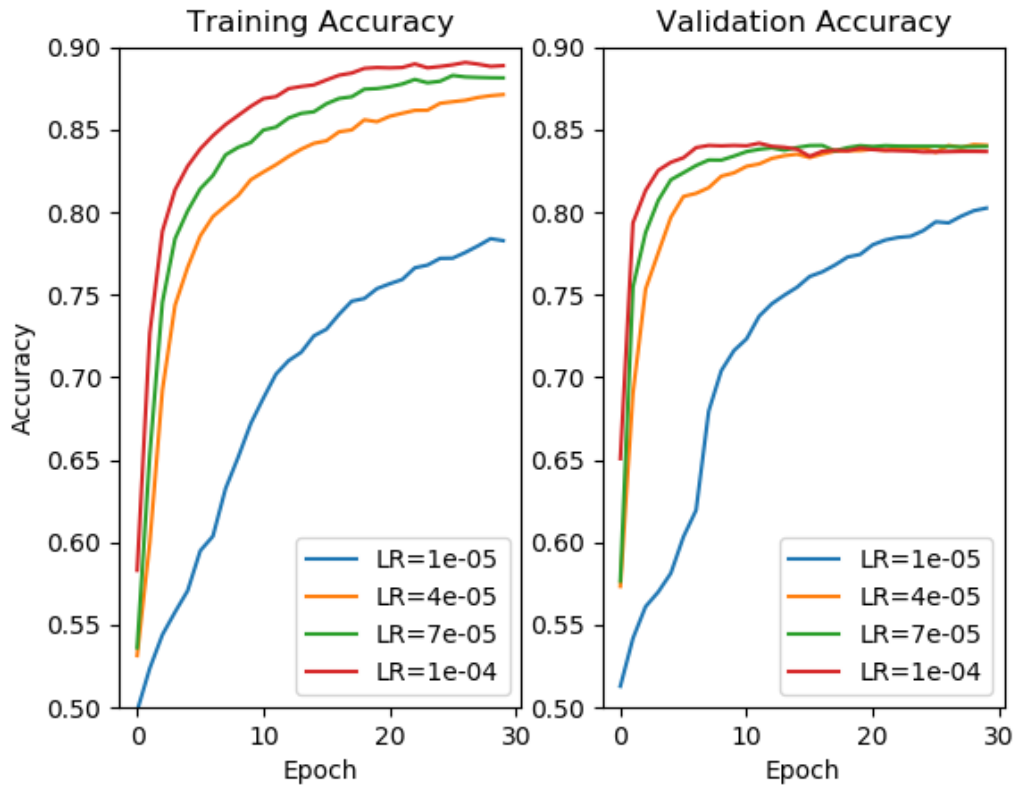
- 大网络的问题
  - 超过两百万个参数，比训练集里的单词还多
  - Embedding层的参数占了95%以上，可能掩盖我们感兴趣的解码器之间的差异
  - 上图结果也表明收敛过快而且过拟合强
- 最终的选择
  - 5000个单词足够包括绝大部分常用词
  - 几万个样本的二分类任务使用64维词向量足矣
  - Embedding层的参数只占80%左右
  - 实现了和大模型一样的正确率

解码器参数选择

- 目标是选取在30个epoch内能达到最佳性能的模型
- 选择初始学习率
  - 根据前期试验，发现学习率影响远大于其他参数
  - 在1e-5到1e-3间调整
- 选择Batch size
  - 在32到256之间
- 选择Dropout
  - 0.1到0.5之间

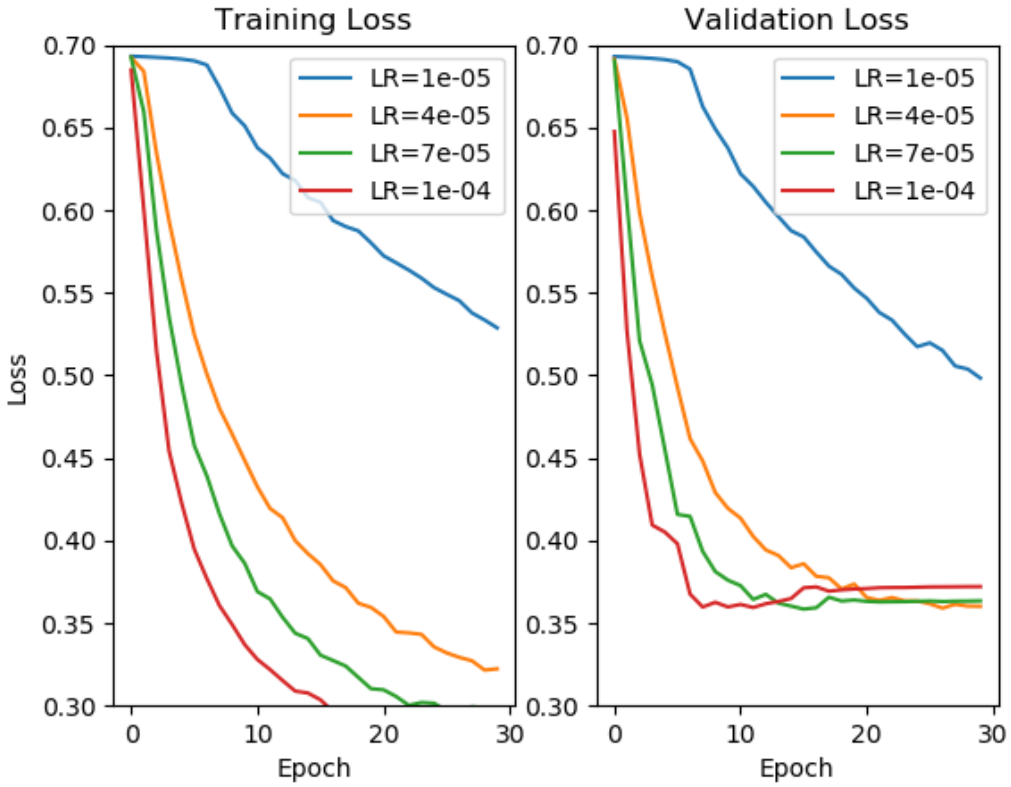
LSTM

- 学习率：4e-5收敛较快且表现较好



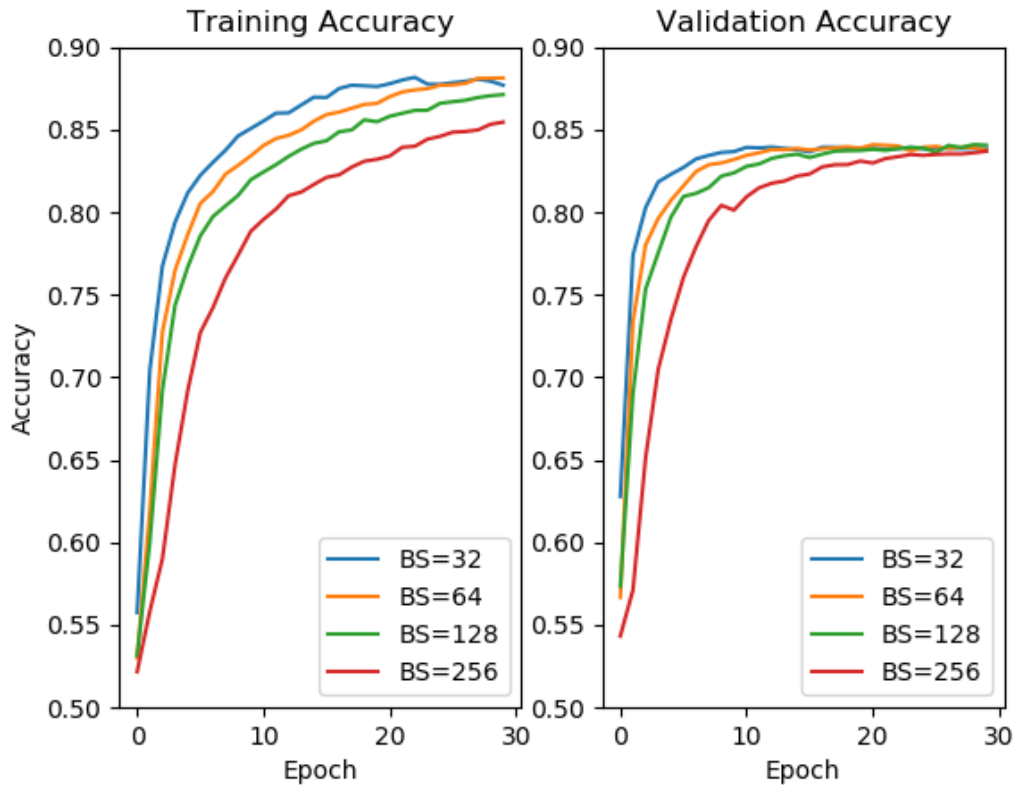
LSTM

- 学习率：4e-5收敛较快且表现较好



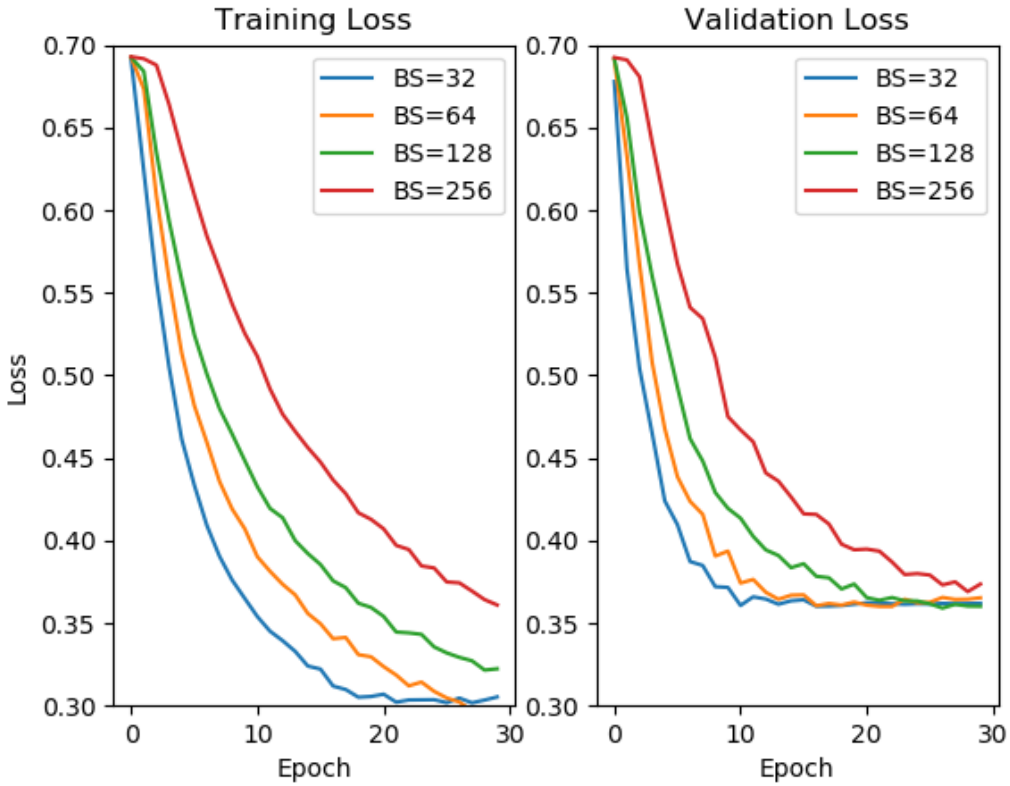
LSTM

- Batch Size: 128表现较好且训练较快



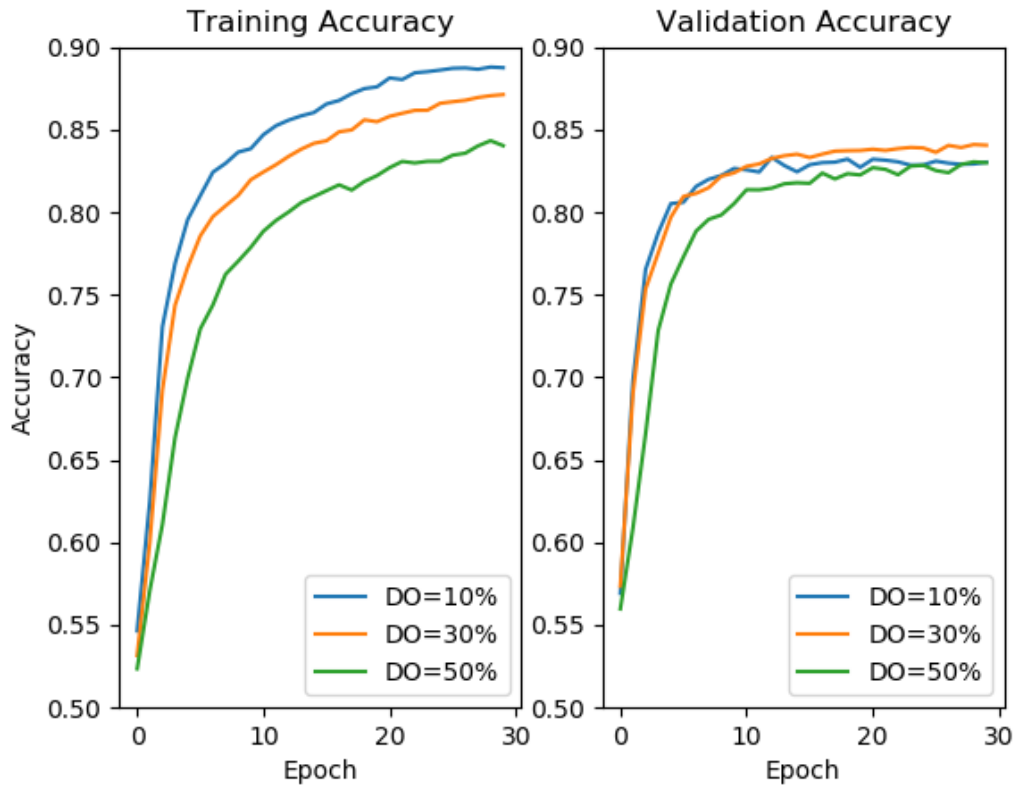
LSTM

- Batch Size: 128表现较好且训练较快



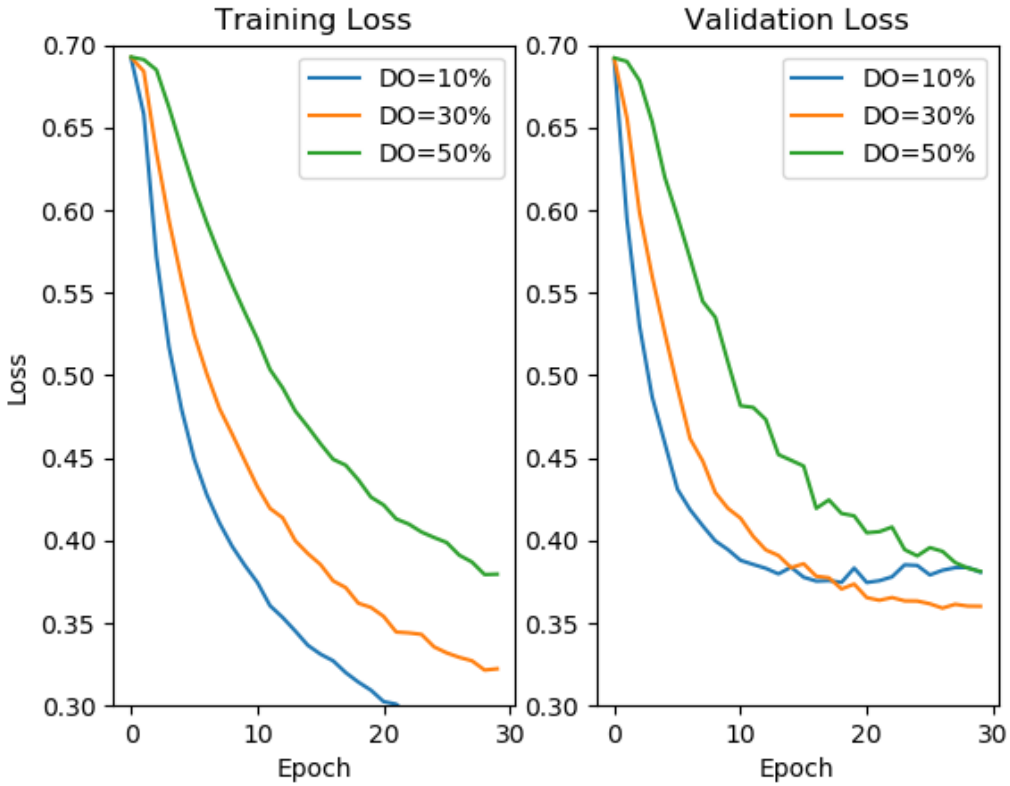
LSTM

- Dropout：30%时表现最好



LSTM

- Dropout：30%时表现最好

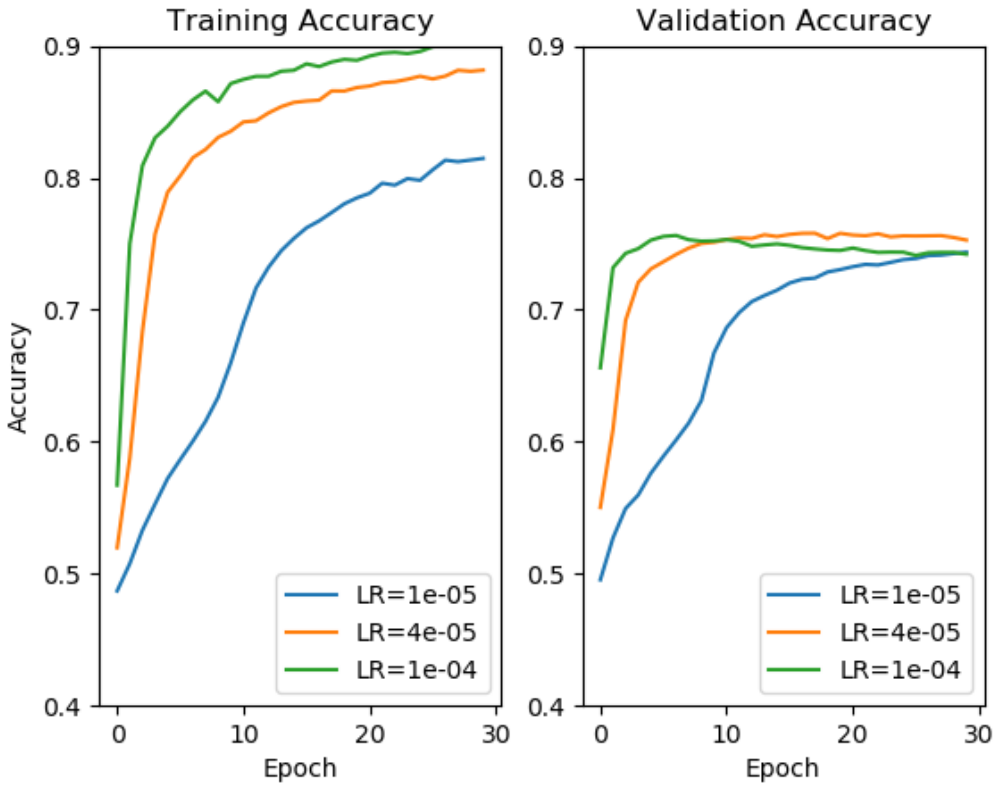


LSTM

- LSTM总体上较容易训练
  - 学习率、batch size、Dropout在很大范围内变动时最终的泛化错误率都基本稳定
  - 即使使用高达50%的Dropout也能平稳收敛

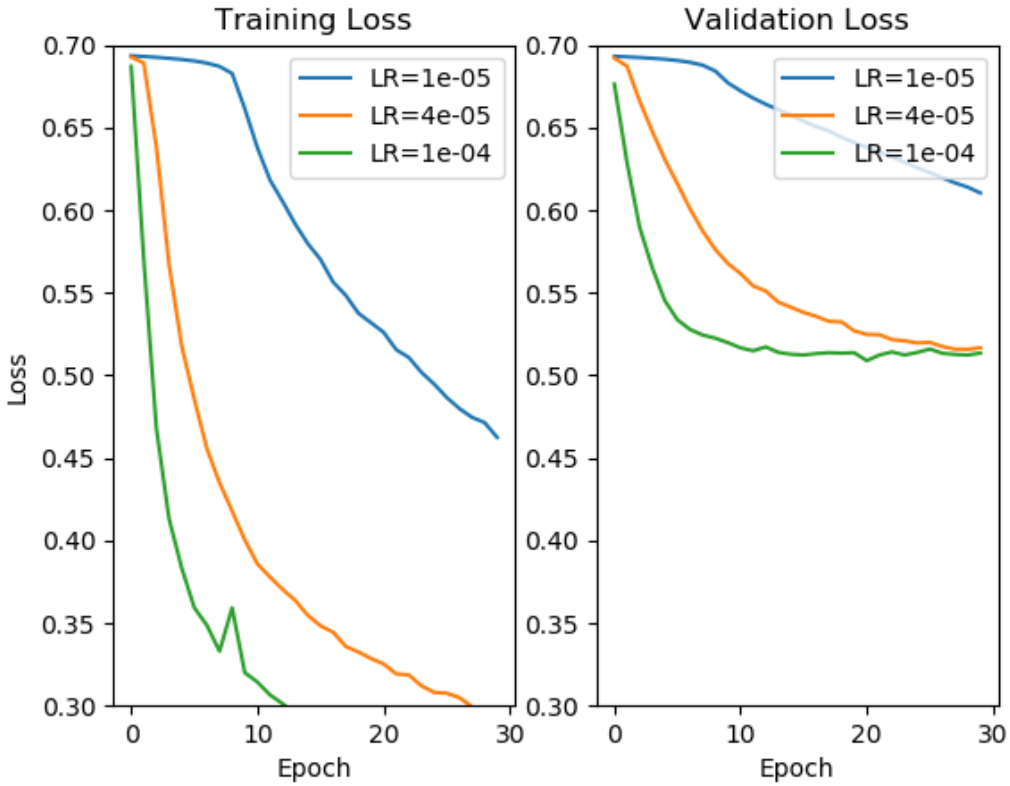
GRU

- 学习率：4e-5收敛较快且表现较好



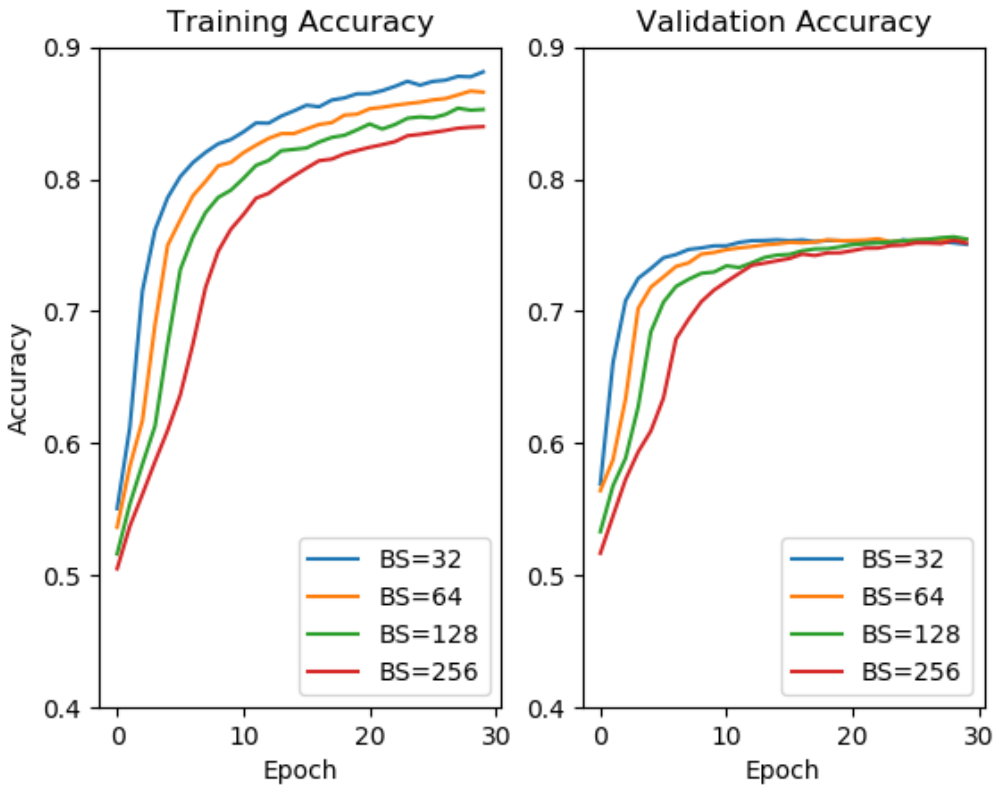
GRU

- 学习率：4e-5收敛较快且表现较好



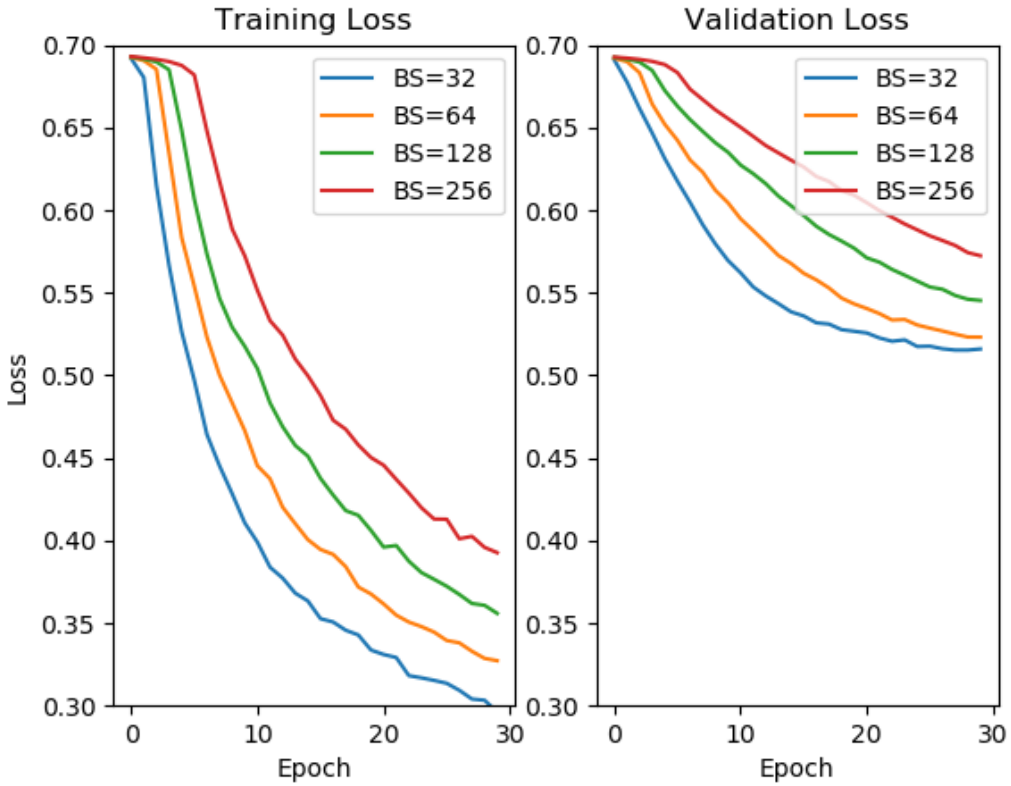
GRU

- Batch Size: 成绩相似，这里选128训练较快



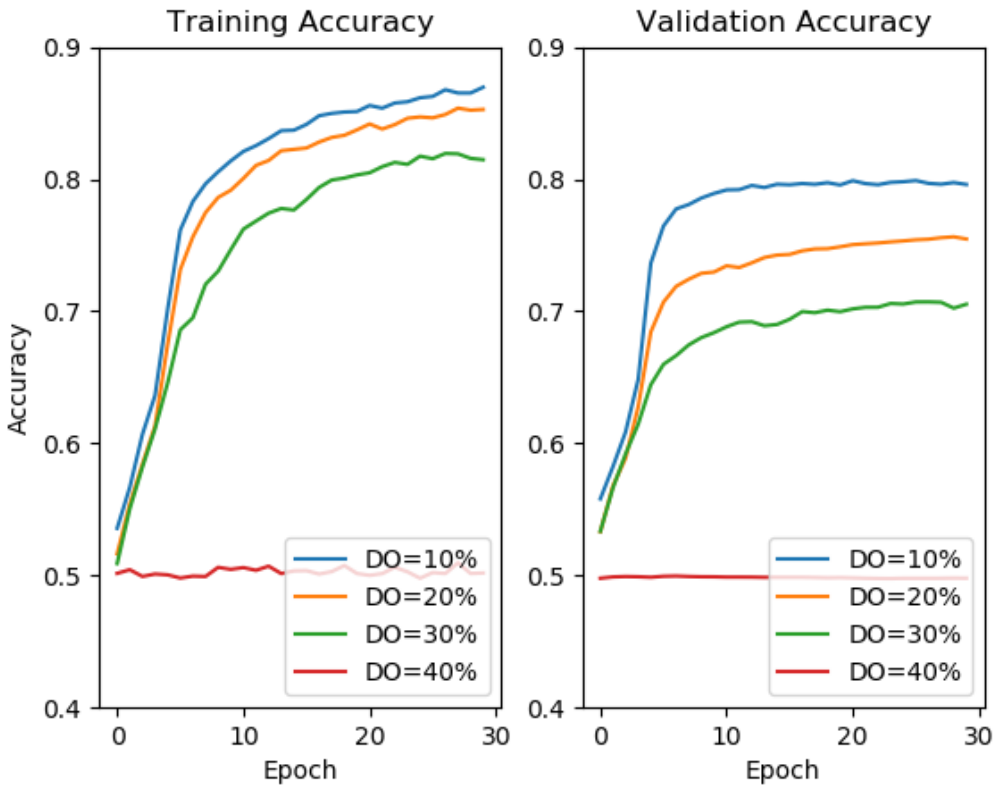
GRU

- Batch Size:成绩相似，这里选128训练较快



GRU

- Dropout：10%时最好，40%时已无法训练



GRU

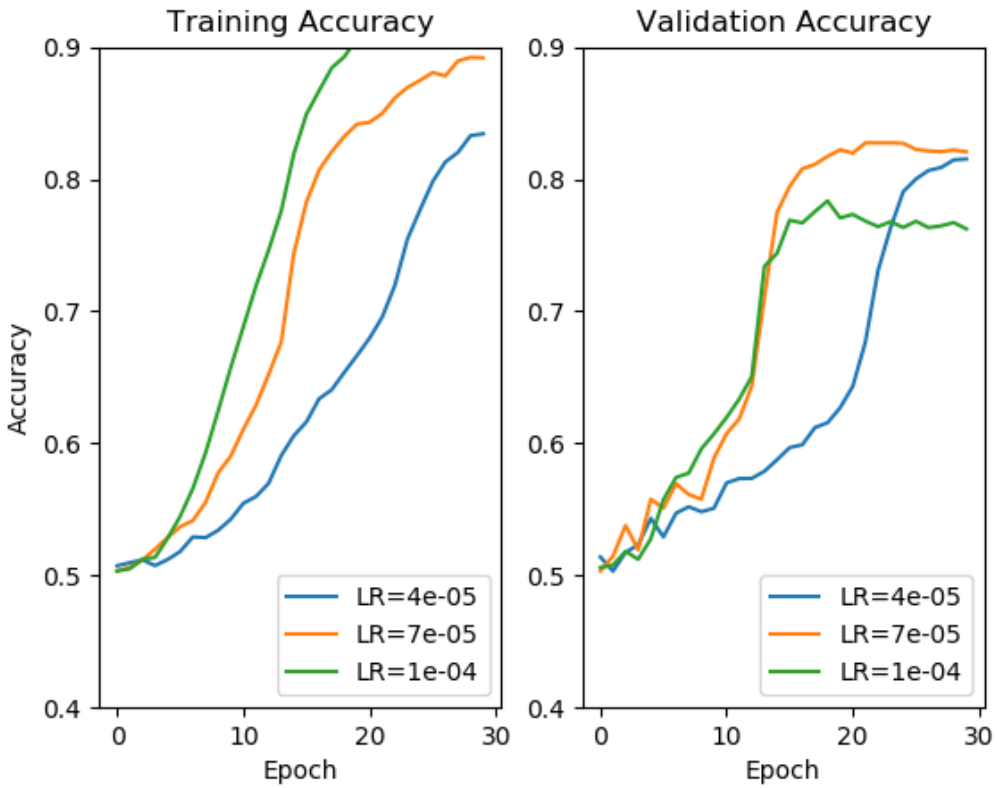
- Dropout：10%时最好，40%时已无法训练

GRU

- 训练快
  - 但过拟合也很快
- 泛化能力不如LSTM
  - 且加大Dropout并不能起到帮助作用
- 不够稳定
  - 只能容忍30%的Dropout
  - 容易出现Loss突然的大幅波动

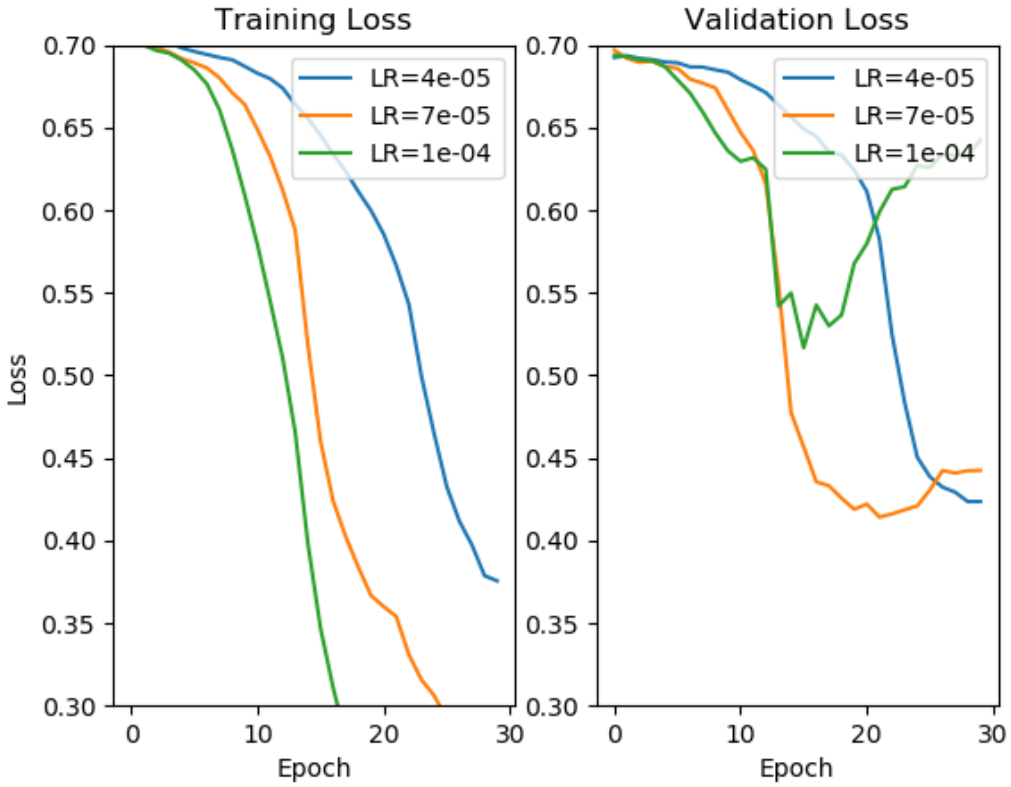
RNN

- 学习率：需要更大学习率如7e-5



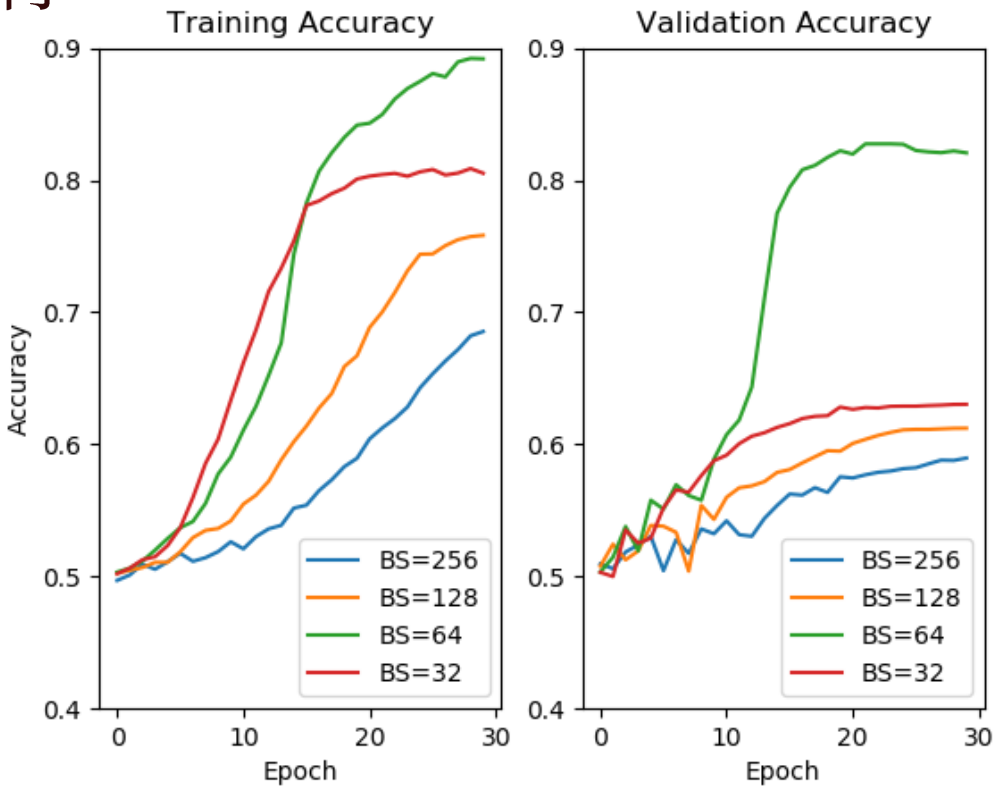
RNN

- 学习率：需要更大学习率如7e-5



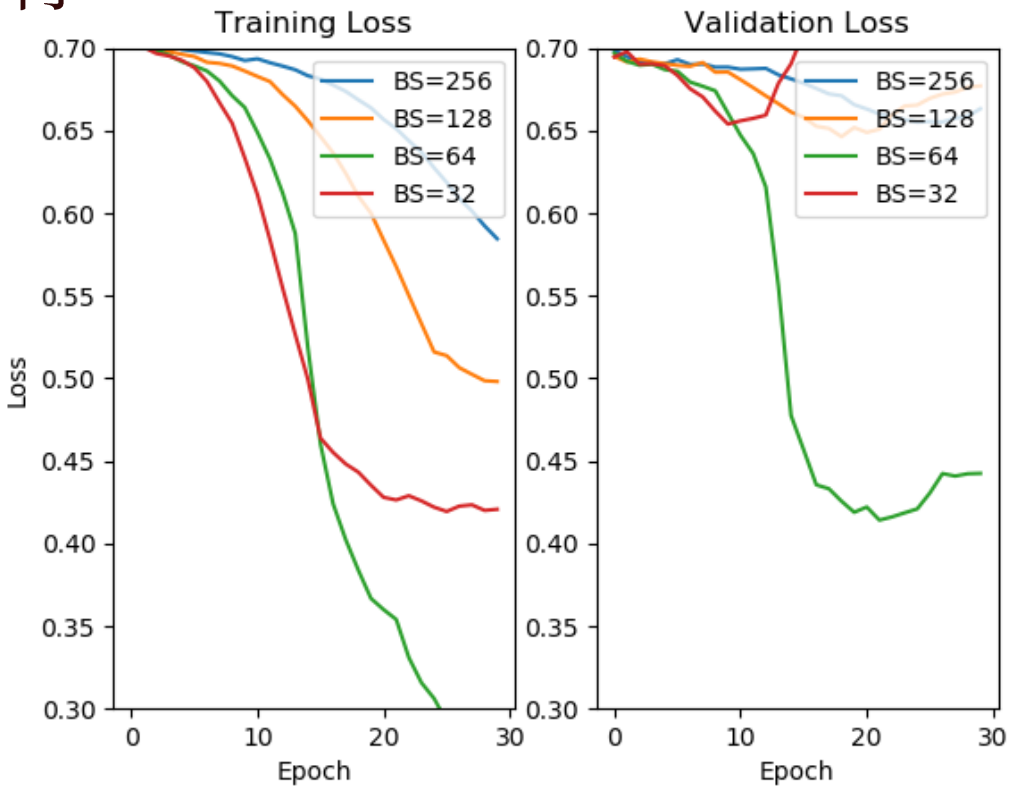
RNN

- Batch size：需要较小batch，并且不是每次都行



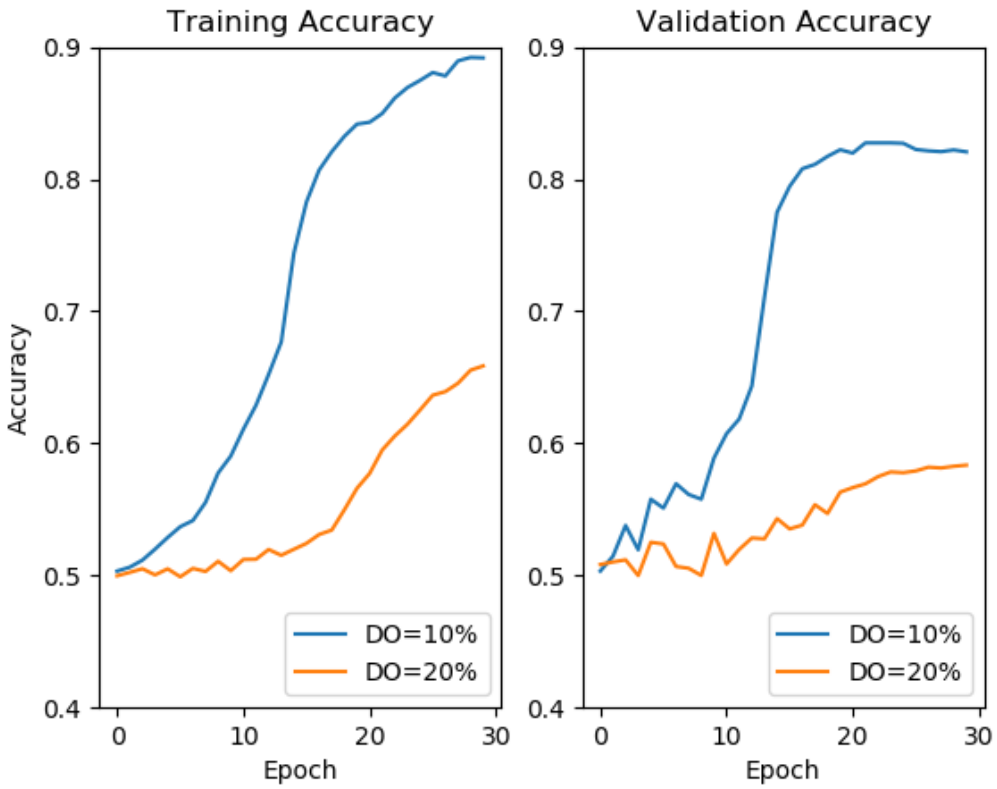
RNN

- Batch size：需要较小batch，并且不是每次都行



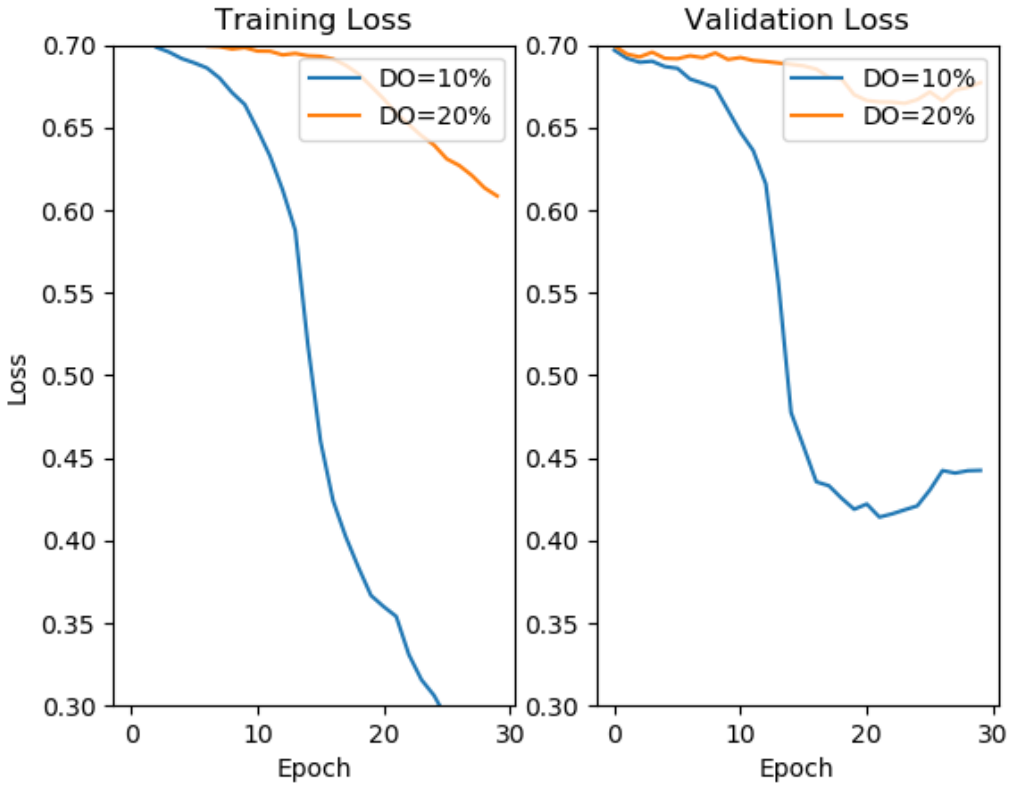
RNN

- Dropout：只能容忍很低的dropout



RNN

- Dropout：只能容忍很低的dropout

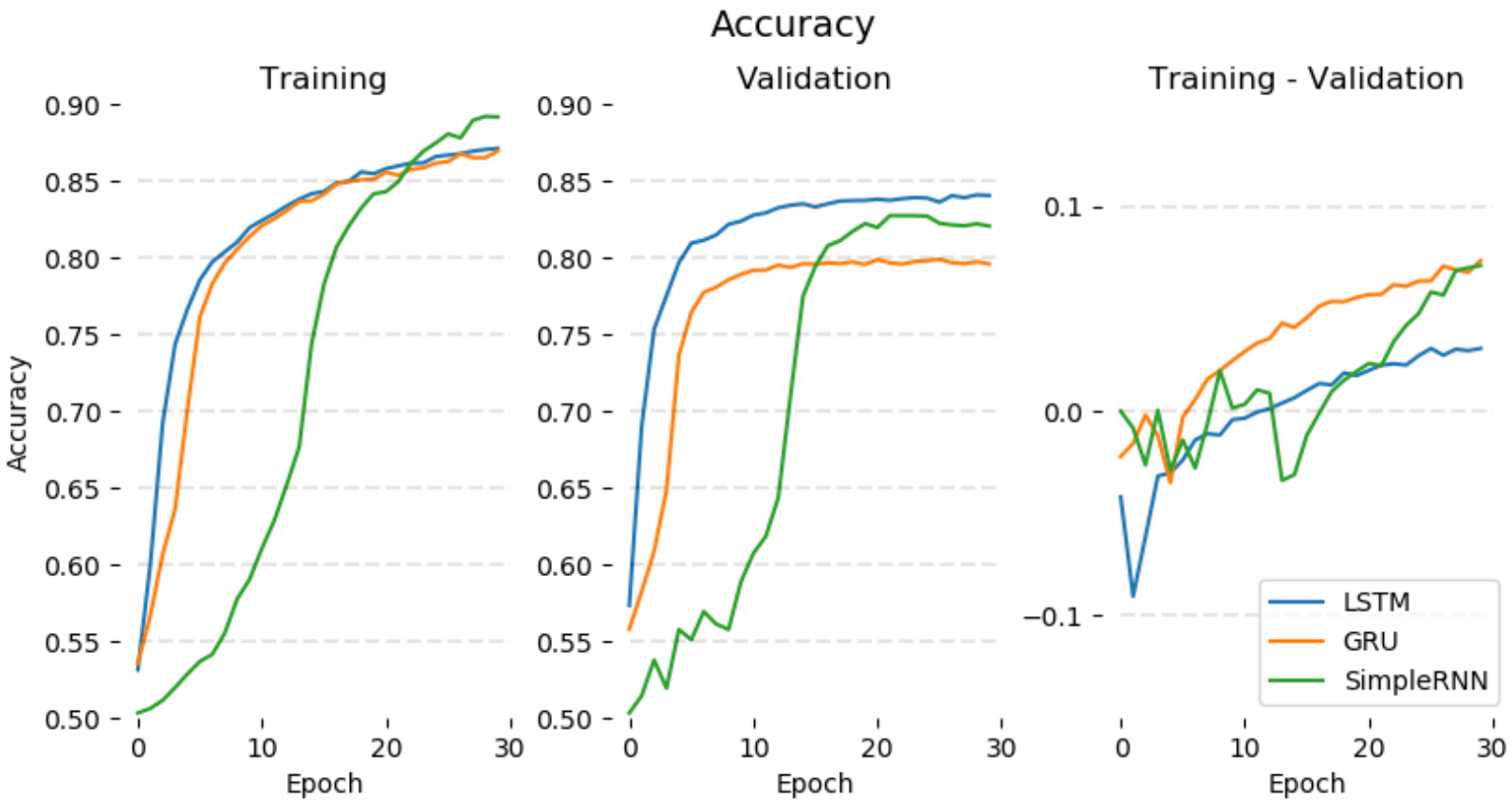


RNN

- 训练非常不稳定
  - 很多次尝试都陷在局部最优出不来
  - Loss下降并非常规的下凸曲线，而是在某个epoch后突然下降，其余基本是线性
  - 需要更小的Batch size和更大的学习率来提供灵活性
  - 只能使用非常低的Dropout
- 运气好时表现非常出色，不比前两种差
  - 运气差时甚至过不了chance level

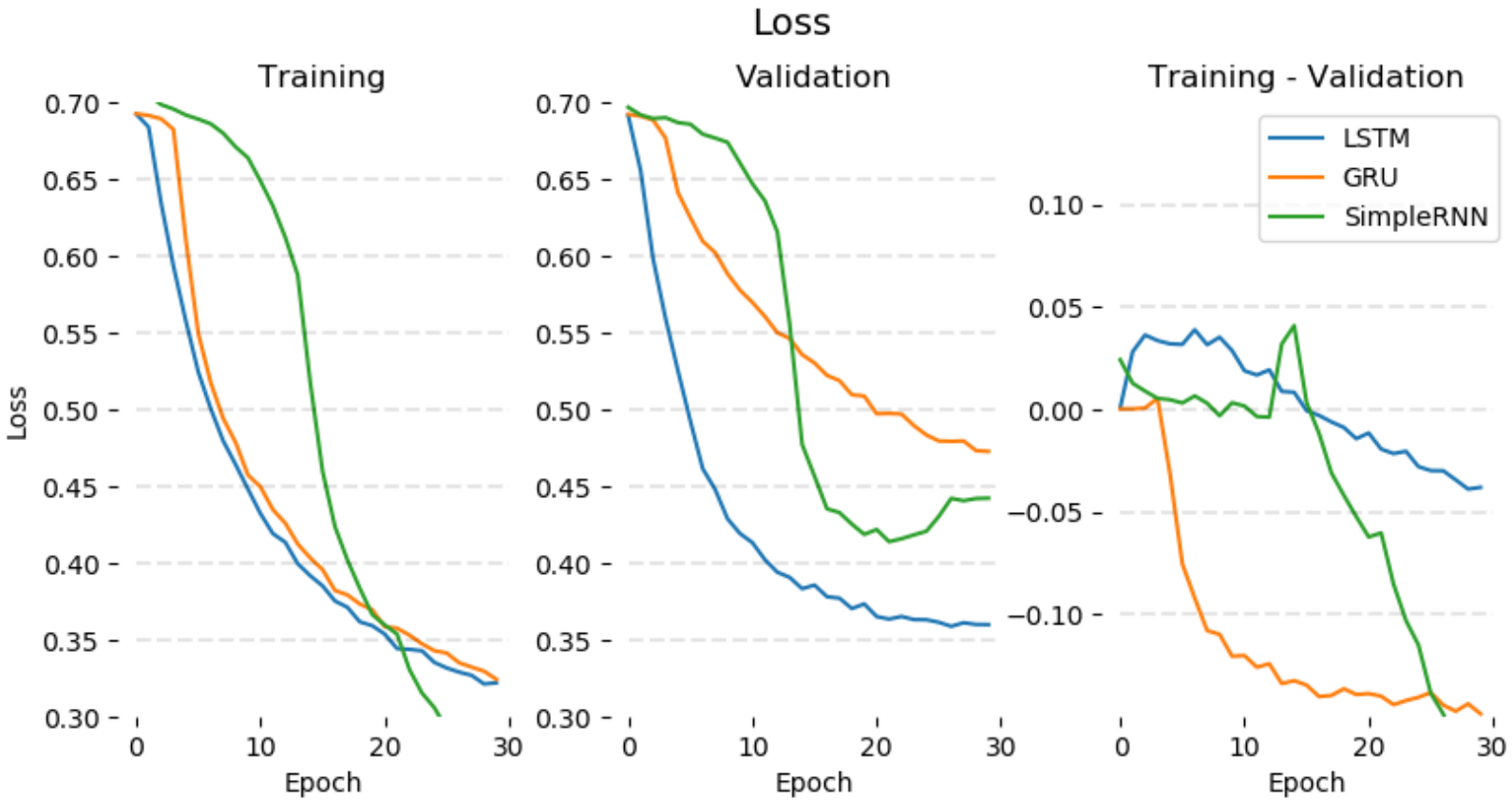
解码器性能比较

- 正确率


Accuracy
Training
Validation
Training - Validation
Accuracy
Epoch
Epoch
Epoch
LSTM
GRU
SimpleRNN

解码器性能比较

❖ Loss

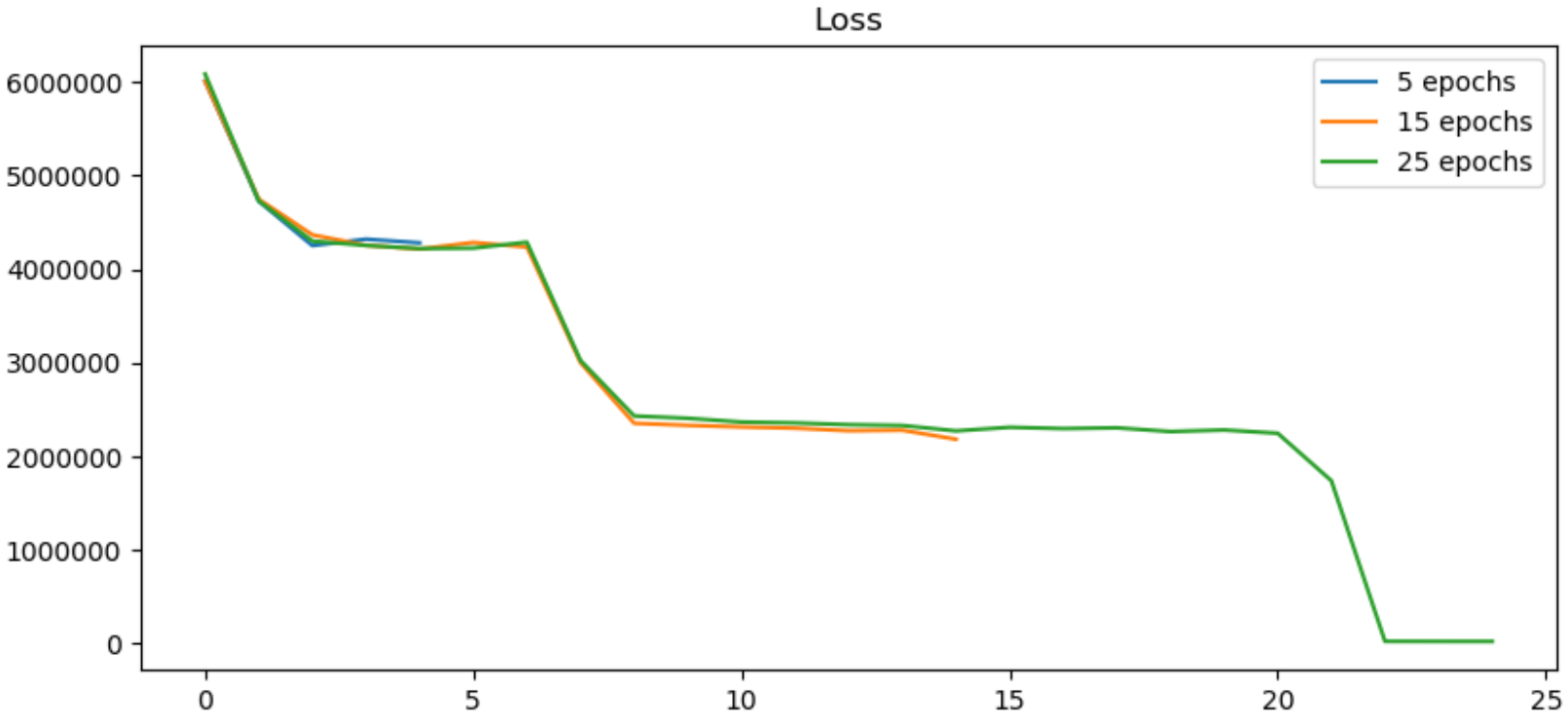


解码器性能比较

- LSTM的综合表现最好
  - 泛化能力最强、训练平稳快速
- GRU的拟合能力强但泛化能力不足
  - 在训练集上表现几乎和LSTM一样好，测试集上却远远不如
  - 训练开始后很快测试集表现就被远远甩开，并且无法用Dropout弥补
- RNN不够稳定
  - Loss下降是阶梯式的而非先快后慢的
  - 容易欠学习又容易过学习

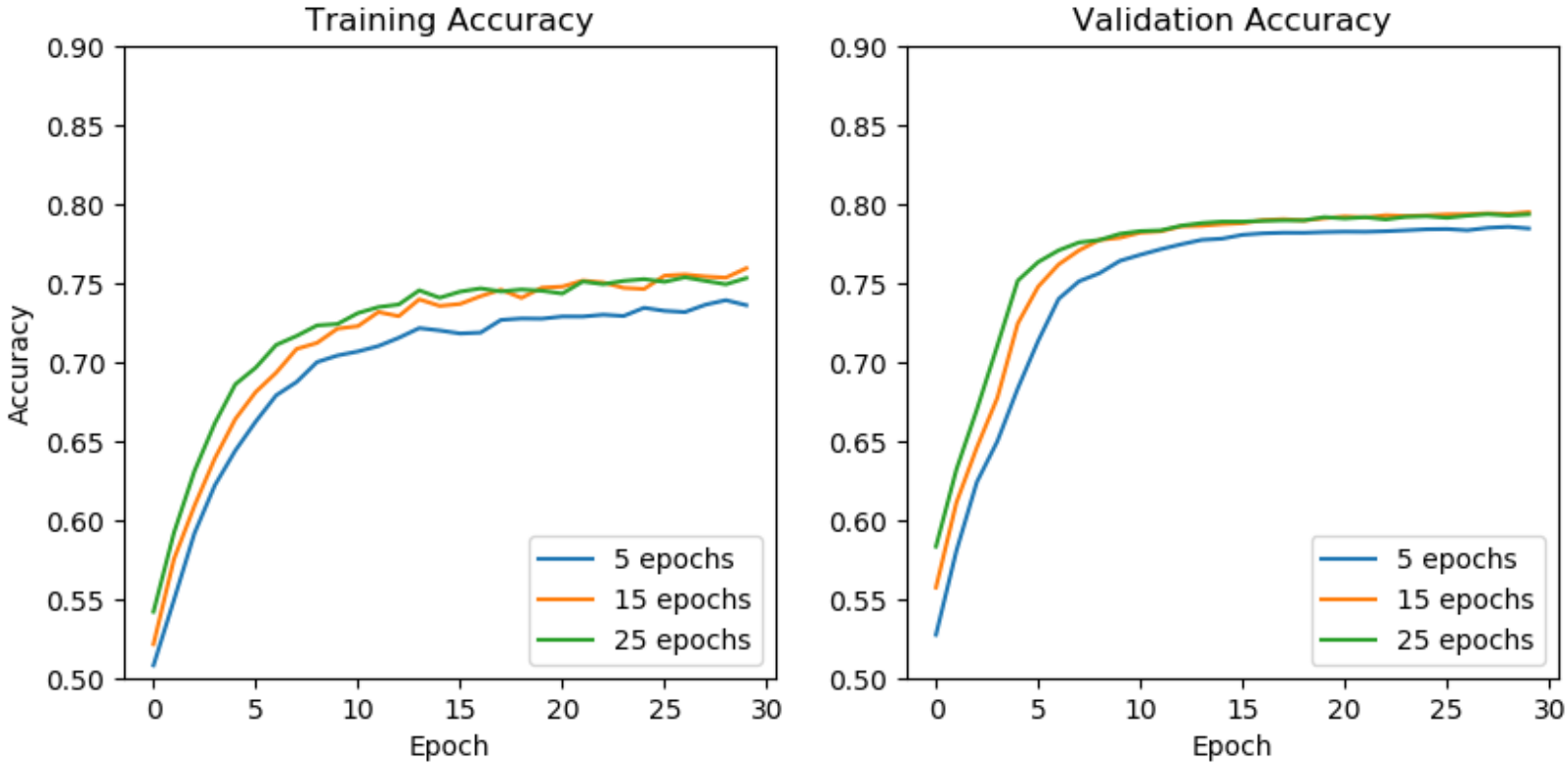
编码器参数选择

- Skip-gram：选取了拟合程度不同的三种词向量用于分类



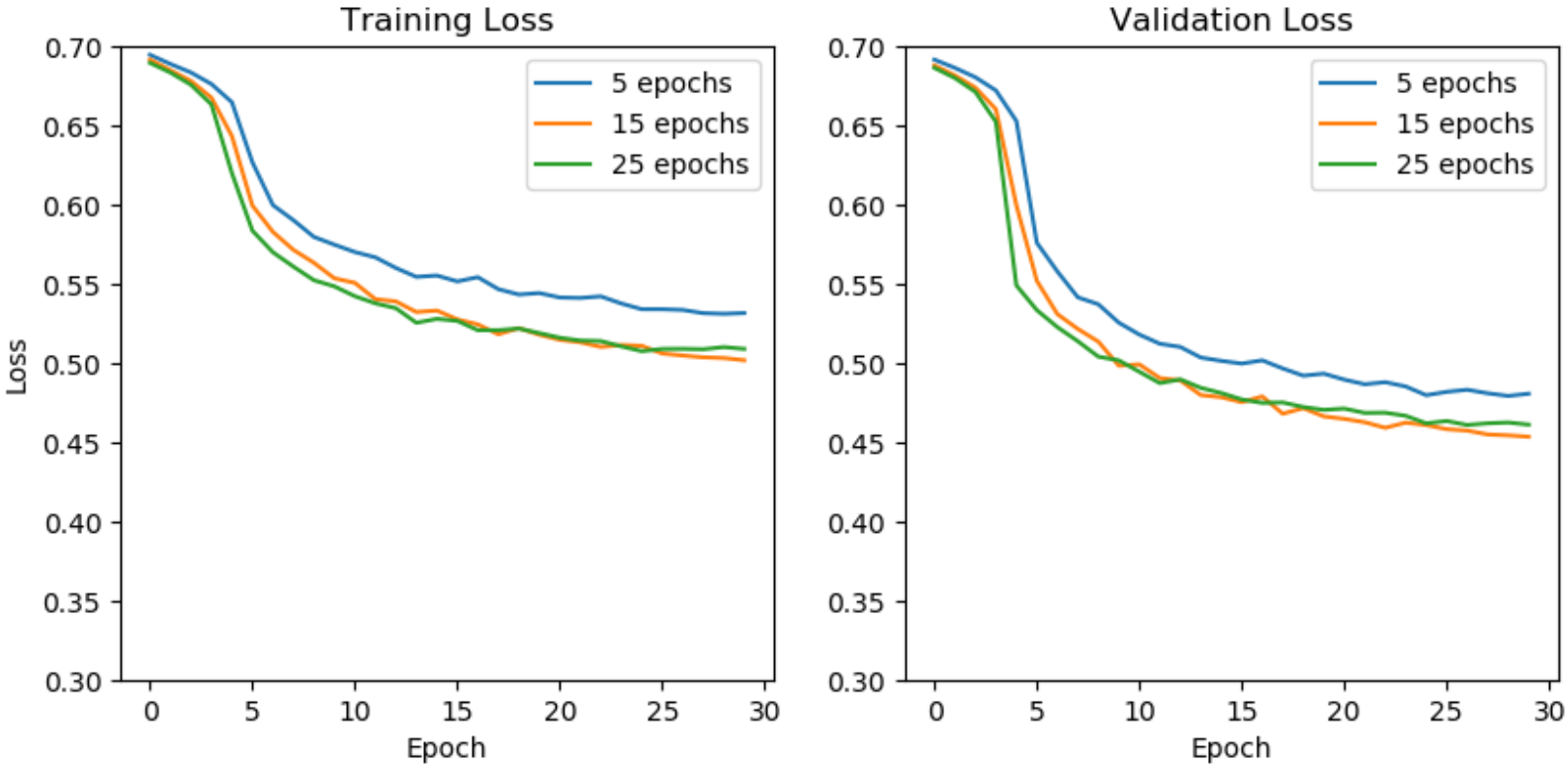
编码器参数选择

- Skip-gram：经过尝试后选定训练参数，使用15轮训练得到的词向量



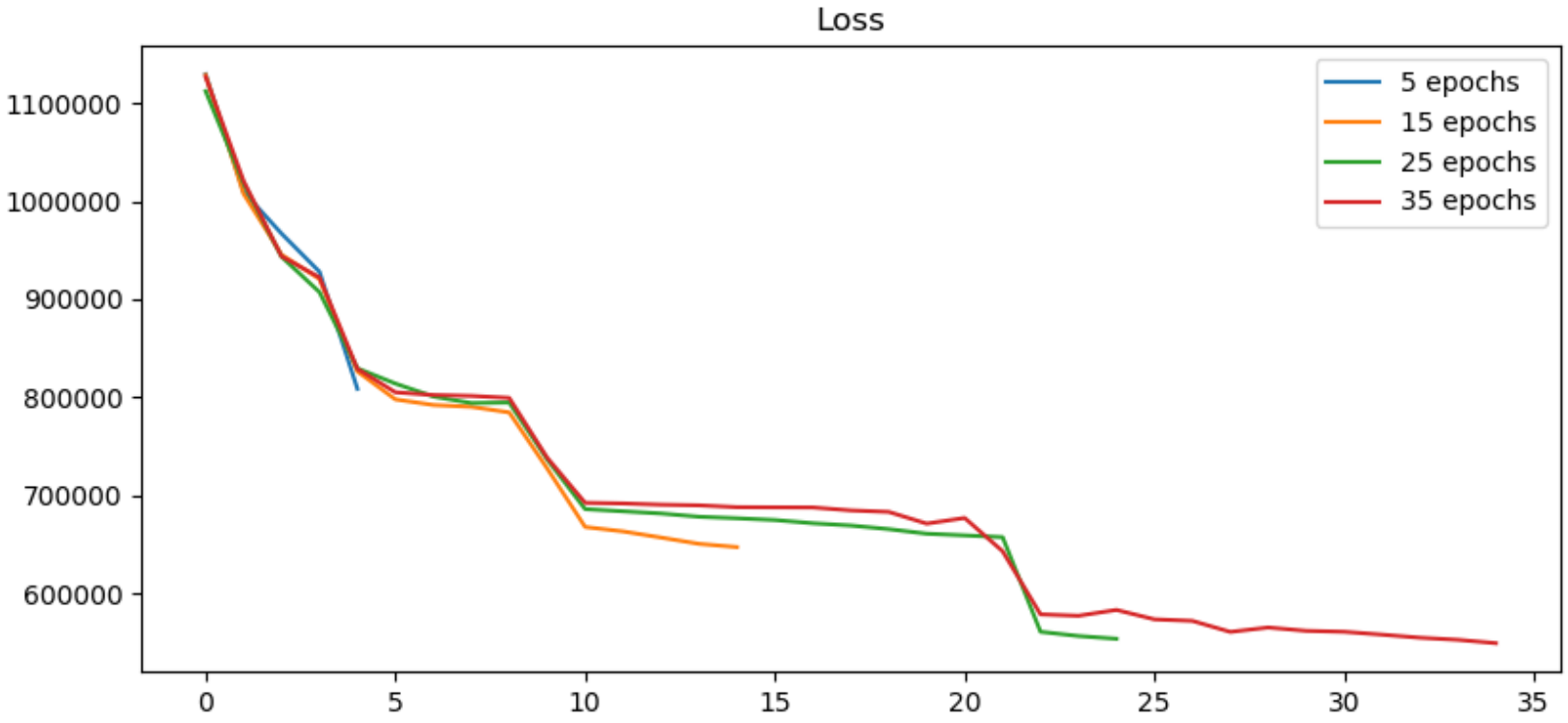
编码器参数选择

- Skip-gram：经过尝试后选定训练参数，使用15轮训练得到的词向量



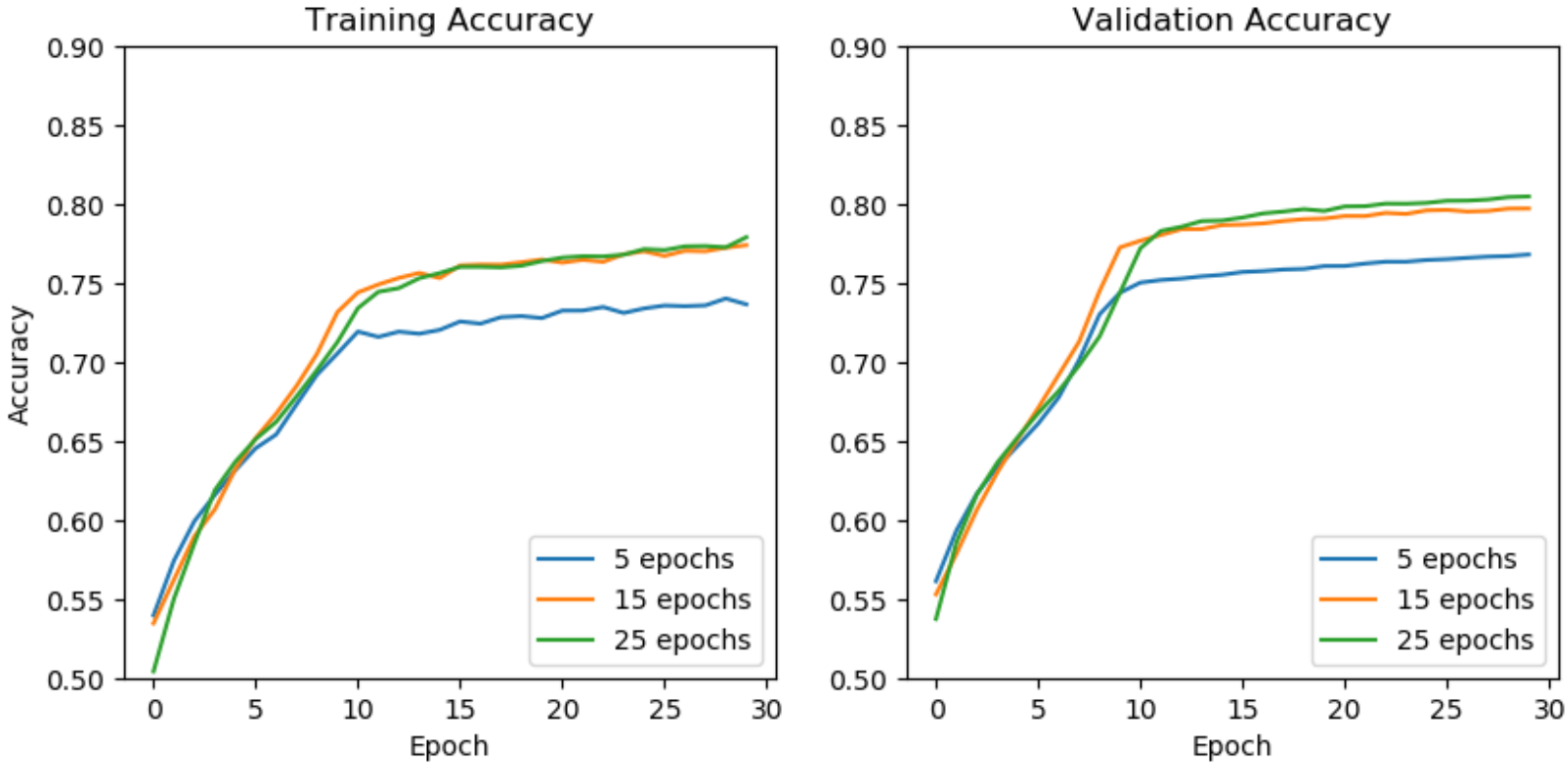
编码器参数选择

- CBOW：同理，选取了训练5/15/25轮得到的词向量用于分类



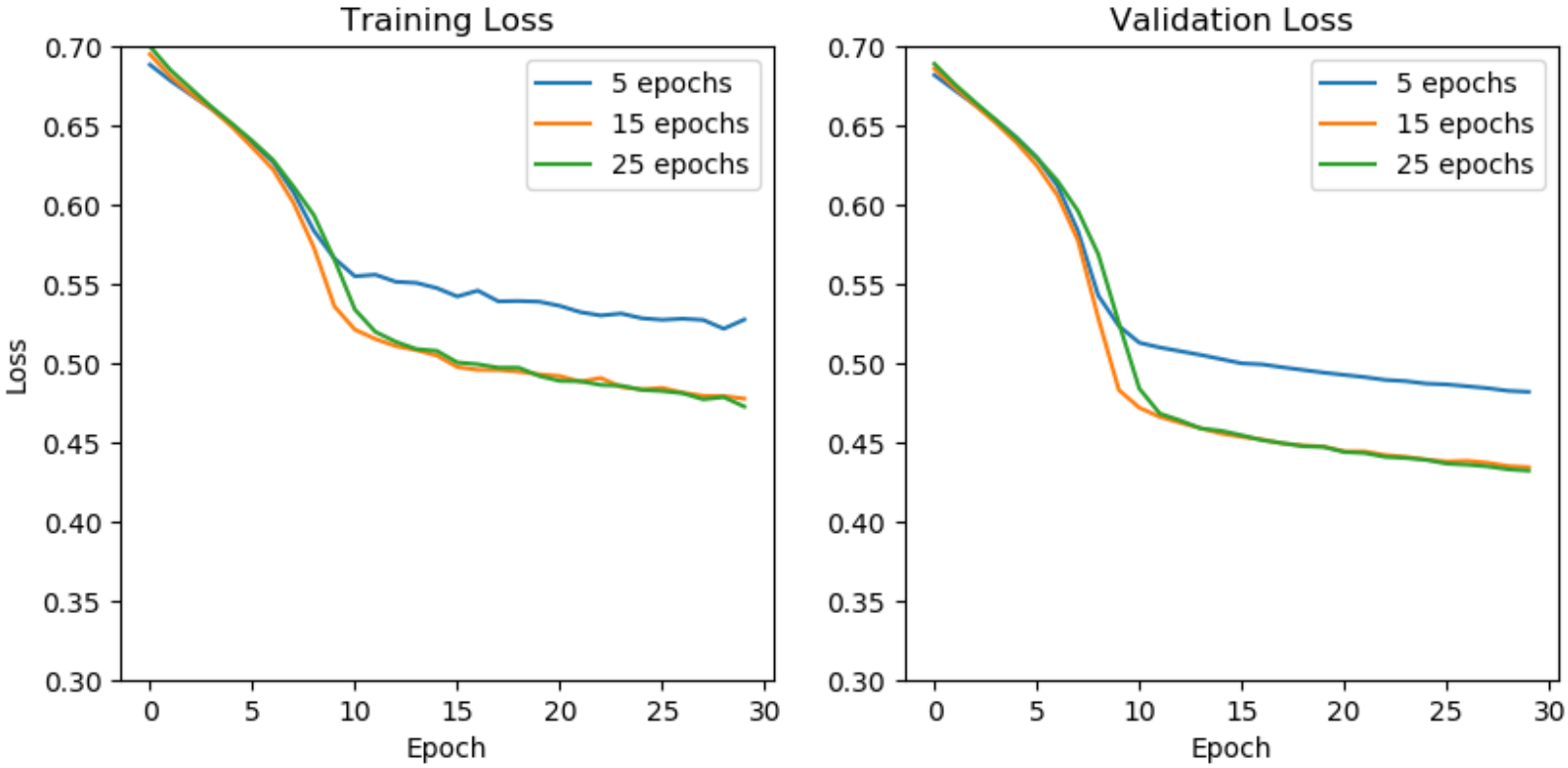
编码器参数选择

- CBOW：经过尝试后选定训练参数，使用25轮训练得到的词向量



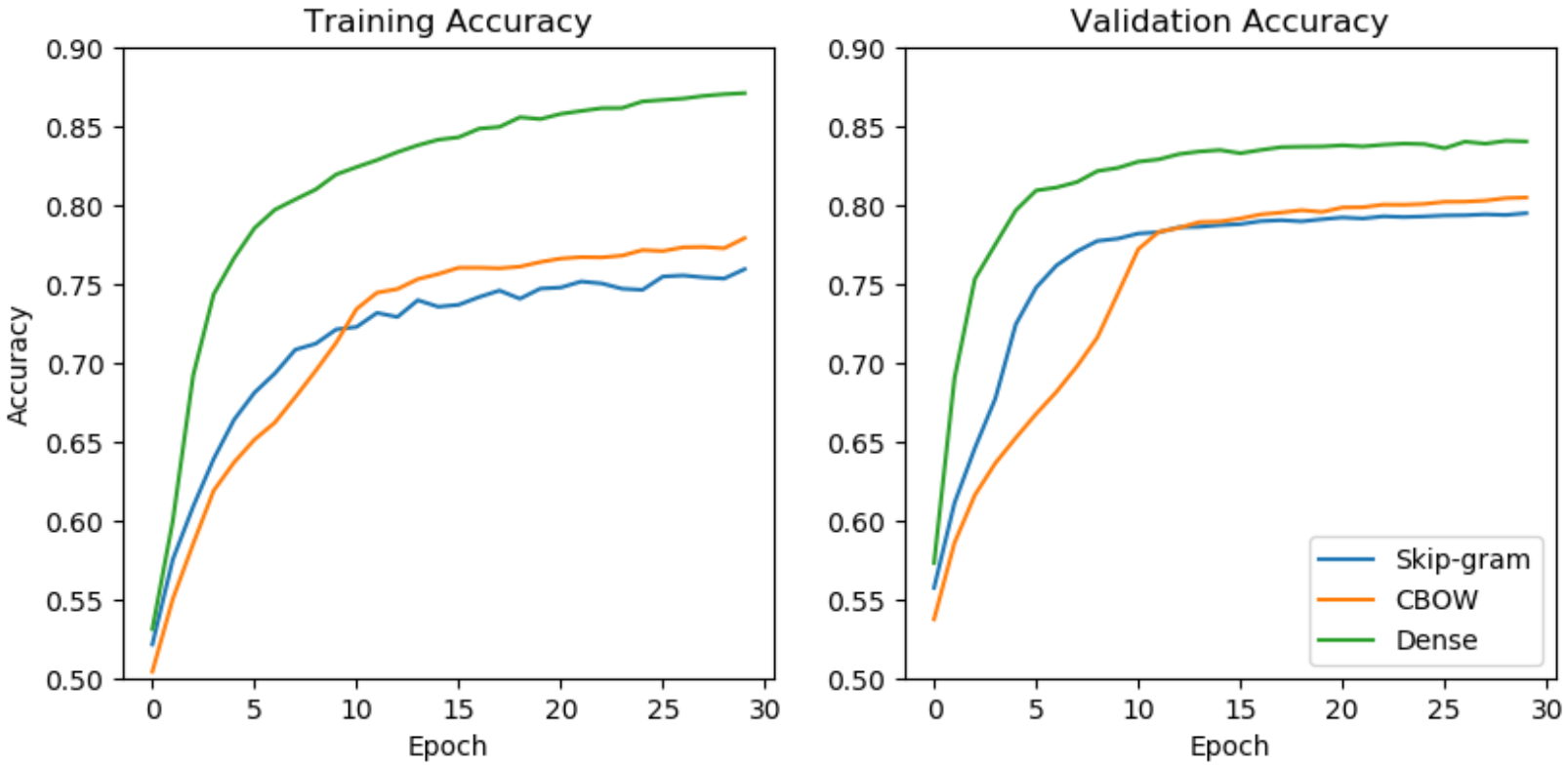
编码器参数选择

- CBOW：经过尝试后选定训练参数，使用25轮训练得到的词向量



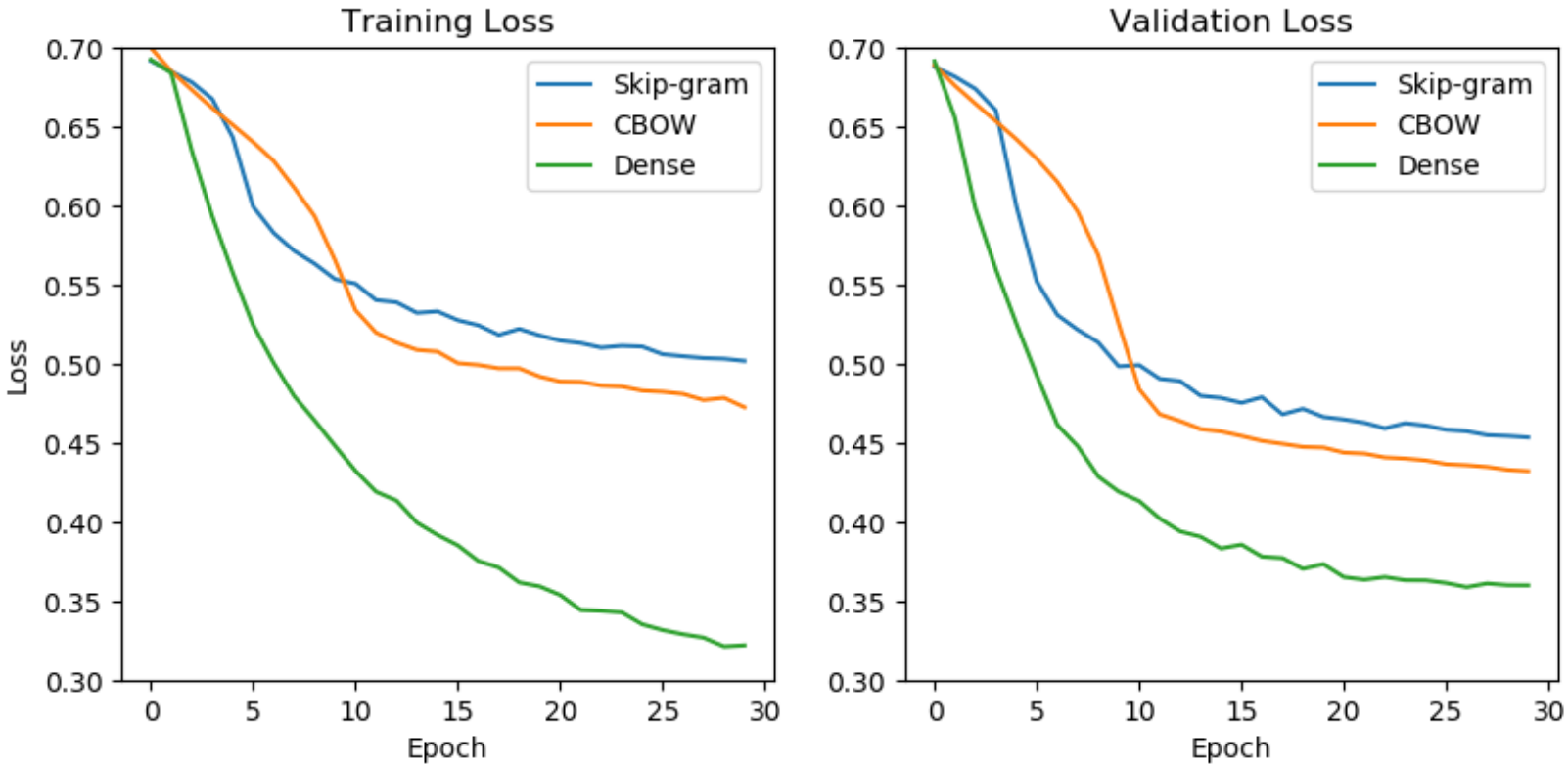
编码器性能比较

- 正确率


Training Accuracy
Validation Accuracy
Accuracy
Epoch
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.55
0.50
0
5
10
15
20
25
30
Skip-gram
CBOW
Dense

编码器性能比较

❖ Loss



编码器性能比较

- 词向量的训练
  - 不同设置差别不大，但是有时仍然会出现欠拟合问题，过拟合似乎不严重
- 编码器的性能
  - 两种无监督方式都不如有监督的共同训练
  - CBOW略微好于Skip-gram

讨论

- 关于分类器的设计
  - Greff, et al. (2015) 研究认为LSTM的不同变体之间并没有性能上的根本差别
  - Jozefowicz, et al. (2015) 发现很多其他RNN架构在一些任务上表现并不比LSTM差
  - 我们的结果显示GRU并没有比其前身LSTM表现更好，甚至有时还没有最简单的RNN好；但是前两者确实在训练的稳定性上远好于RNN
- 关于词向量的编码
  - 不引入外部语义时，直接与分类器一起训练或许能得到最好的词向量编码

讨论

- 可以改进的方向：
  - 我们使用的网络结构和数据集都比较小，这也是造成GRU和LSTM无法发挥的重要原因
  - 我们以30个epoch内的成绩判断分类表现，可能造成训练比较粗糙，没有找到更好的解
  - 可以在每个配置下多运行几次以获得统计上的结论
  - 可以考虑纳入其他非神经网络的模型进行比较