机器学习保险行业问答开放数据集

Hai Liang Wang

2017-08-12

机器学习保险行业问答开放数据集：2．使用案例

目前，使用机器学习解决复杂问题，尤其是从海量数据中，通过深度学习的方式提取特征，然后构建出能识别、分类、标注等新的计算服务成为了大家积极学习和努力探索的方向。图像的风格化、视频的特效道具、智能音箱等各种基于机器学习的产品开始问世，我相信机器学习会在文本方面取得更大的突破，出于学习和交流的目的，我制作了机器学习保险行业问答开放数据集[1]，该语料库是基于优秀的英语语料翻译而来，经过对比，该语料可以用于研究和学习，从规模和质量上，是目前中文问答语料中，保险行业垂直领域最优秀的语料，关于该语料制作过程和结构设计，通过语料主页了解。

DeepQA-1

为了展示如何使用该语料训练模型和评测算法，我做了一个示例项目 - DeepQA-1[2]，本文接下来会介绍 DeepQA-1，假设读者了解深度学习基本概念和 Python 语言。

Data Loader

数据加载包含两部分：加载语料和预处理。加载数据使用 insuranceqa_data [3] 载入训练，测试和验证集的数据。

Figure 1: 安装 insuranceqa_data

```
$ pip install insuranceqa_data
Collecting insuranceqa_data
  Downloading insuranceqa_data-2.1.tar.gz
Building wheels for collected packages: insuranceqa-data
  Running setup.py bdist_wheel for insuranceqa-data ... done
  Stored in directory: /Users/hain/Library/Caches/pip/wheels/5
e744582e749dad01fb0409ca0e886619bea58bea
Successfully built insuranceqa-data
Installing collected packages: insuranceqa-data
Successfully installed insuranceqa-data-2.1
```

[1] insuranceqa-corpus-zh, https://github.com/Samurais/insuranceqa-corpus-zh
[2] deep_qa_1, https://github.com/Samurais/insuranceqa-corpus-zh
[3] insuranceqa_data, http://alturl.com/bae3q

Figure 2: 加载问答对数据

```
import insuranceqa_data as insuranceqa

_train_data = insuranceqa.load_pairs_train()
_test_data = insuranceqa.load_pairs_test()
_valid_data = insuranceqa.load_pairs_valid()
```

预处理是按照模型的超参数处理问题和答案，将它们组合成输入需要的格式，在本文介绍的 baseline model 中，预处理包含下面工作：

1. 在词汇表 (vocab) 中添加辅助 Token: <PAD>, <GO>. 假设 q 是问题序列，u 是回复序列，输入序列可以表示为：

$$(q_1, ..., q_{question\_max\_length}, < GO >, u_1, ..., u_{utterance\_max\_length}) \quad (1)$$

超参数question_max_length代表模型中问题的最大长度。超参数utterance_max_length代表模型中回复的最大长度，回复可能是正例，也可能是负例。

Figure 3: 加载词汇表

```
vocab_data = insuranceqa.load_pairs_vocab()
print("keys", vocab_data.keys())
vocab_size = len(vocab_data['word2id'].keys())
VOCAB_PAD_ID = vocab_size+1
VOCAB_GO_ID = vocab_size+2
vocab_data['word2id']['<PAD>'] = VOCAB_PAD_ID
vocab_data['word2id']['<GO>'] = VOCAB_GO_ID
vocab_data['id2word'][VOCAB_PAD_ID] = '<PAD>'
vocab_data['id2word'][VOCAB_GO_ID] = '<GO>'
```

其中，Token <GO> 用来分隔问题和回复，Token <PAD> 用来补齐问题或回复。

Figure 4: 构建输入序列

```
def pack_question_n_utterance(q, u, q_length = 20, u_length = 99):
    '''
    combine question and utterance as input data for feed-forward network
    '''
    assert len(q) > 0 and len(u) > 0, "question and utterance must not be empty"
    q = padding(q, VOCAB_PAD_ID, q_length)
    u = padding(u, VOCAB_PAD_ID, u_length)
    assert len(q) == q_length, "question should be pad to q_length"
    assert len(u) == u_length, "utterance should be pad to u_length"
    return q + [VOCAB_GO_ID] + u
```

训练数据 $\_train\_data$(图 2) 包含了 141,779 条，正例: 负例 =1:10，根据超参数 $batch\_size$ 生成输入序列:

Figure 5: 构建输入序列

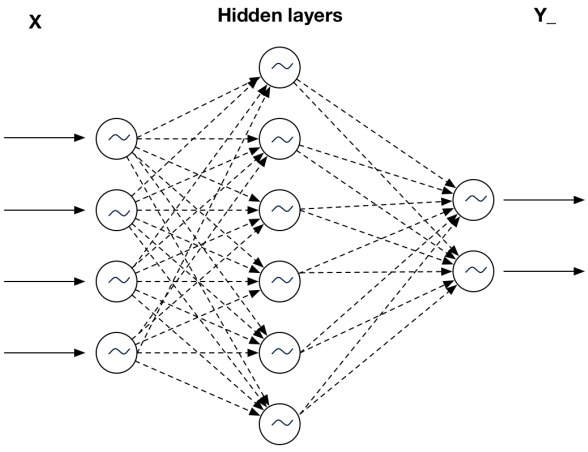
```
batch_iter = BatchIter(data = data, batch_size = batch_size)

for mini_batch in batch_iter.next():
    result = []
    for o in mini_batch:
        x = pack_question_n_utterance(o['question'], o['utterance'], question_max_length, utterance_max_length)
        y_ = o['label']
        assert len(x) == utterance_max_length + question_max_length + 1, "Wrong length afer padding"
        assert VOCAB_GO_ID in x, "<GO> must be in input x"
        assert len(y_) == 2, "desired output."
        result.append([x, y_])
    if len(result) > 0:
        # print('data in batch:%d' % len(mini_batch))
        yield result
    else:
        raise StopIteration
```

图 5 中 x 就是输入序列 (公式 1)。y 代表标注数据: 正例还是负例，正例标为 [1,0]，负例标为 [0,1]，这样做的好处是方便计算损失函数和准确度。测试数据和验证数据也用同样的方式进行处理，唯一不同的是它们不需要做成 mini-batch。需要强调的是，处理词汇表和构建输入序列的方式可以尝试用不同的方法，上述方案仅作为表达 baseline 结果而采用，一些有助于增强模型能力的，比如使用 word2vec 训练词向量，设计输入序列为 (Q, U_{pos}, U_{neg}) 等都值得尝试。

Network

baseline model 使用了最简单的神经网络，输入序列 X 从左侧进入，输出序列 $Y\_$ 输出包含 2 个数值的 vector，然后使用损失函数计算误差。

Figure 6: 神经网络



超参数，**Hyper params**

Figure 7: 超参数列表

```
class NeuralNetwork():
    def __init__(self, hidden_layers = [100, 50],
                 question_max_length = 20,
                 utterance_max_length = 99,
                 lr = 0.001, epoch = 10,
                 batch_size = 100,
                 eval_every_N_steps = 500):
        '''
        Neural Network to train question and answering model
        '''
        self.input_layer_size = question_max_length + utterance_max_length + 1 # 1 is for <GO>
        self.output_layer_size = 2 # just the same shape as labels
        self.layers = [self.input_layer_size] + hidden_layers + [self.output_layer_size] # [2] is for output layer
        self.layers_num = len(self.layers)
        self.weights = [np.random.randn(y, x) for x,y in zip(self.layers[:-1], self.layers[1:])]
        self.biases = [np.random.randn(x, 1) for x in self.layers[1:]]
        self.epoch = epoch
        self.lr = lr
        self.batch_size = batch_size
        self.eval_every_N_steps = eval_every_N_steps
        self.test_data = corpus.load_test()
```

参数项	描述	默认值
hidden_layers	隐含层，比如 [100,50] 代表两个隐含层，分别有 100，50 个神经元	[100, 50]
question_max_length	问题的最大长度	20
utterance_max_length	回复的最大长度	99
lr	学习率	0.001
batch_size	每次根据随机梯度下降理论计算误差的批处理大小	100
eval_every_N_steps	每迭代 N 次后使用测试集评测准确度	500

Table 1: 超参数

损失函数

神经网络的激活函数使用 *sigmoid* 函数，损失函数使用最大似然的思想。

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log h_\theta(x^{(i)}) + (1-y^{(i)})\log(1-h_\theta(x^{(i)}))\right]$$

Figure 8: 损失函数

```
def loss_fn(self, a, y_):
    # cross-entropy cost fn
    return np.sum(np.nan_to_num(-y_*np.log(a) - (1-y_)*np.log(1-a)))
```

迭代训练

使用 mini-batch 加载数据，迭代训练的大部分工作在 *back_propagation* 中完成，*back_propagation* 计算出每次迭代的损失和 b, W 的误差率，然后使用学习率和误差率更新每个 b, W。

```
def run(self, test = False):
    '''
    Train model with mini-batch stochastic gradient descent.
    '''
    total_step = 0
    for n in range(self.epoch):
        for mini_batch in corpus.load_train():
            nabla_b = [np.zeros(b.shape) for b in self.biases]
            nabla_w = [np.zeros(w.shape) for w in self.weights]
            total_cost = 0.0
            for x, y_ in mini_batch:
                # here scale the input's word ids with 0.001 for x to make sure the Z-vector can pass sigmoid fn
                delta_nabla_b, delta_nabla_w, cost = self.back_propagation( \
                            np.reshape(x, (self.input_layer_size, 1)) * 0.001, \
                            np.reshape(y_, (self.output_layer_size, 1)))
                nabla_b = [ nb+mnb for nb, mnb in zip(nabla_b, delta_nabla_b)]
                nabla_w = [ nw+mnw for nw, mnw in zip(nabla_w, delta_nabla_w)]
                total_cost += cost
            self.weights = [ w - (self.lr * w_)/len(mini_batch) for w, w_ in zip(self.weights, nabla_w)]
            self.biases = [ b - (self.lr * b_)/len(mini_batch) for b, b_ in zip(self.biases, nabla_b)]
            total_step += 1
            print("Epoch %s, total step %d, cost %f" % (n, total_step, total_cost/len(mini_batch)))
            visual_loss.plot(total_step, total_cost/len(mini_batch))
            if (total_step % self.eval_every_N_steps ) == 0 and test:
                accuracy = self.evaluate()
                print("Epoch %s, total step %d, accuracy %s" % (n, total_step, accuracy))
                visual_acc.plot(total_step, accuracy)
```

Listing 1: 运行训练脚本

```
python3 deep_qa_1/network.py
```

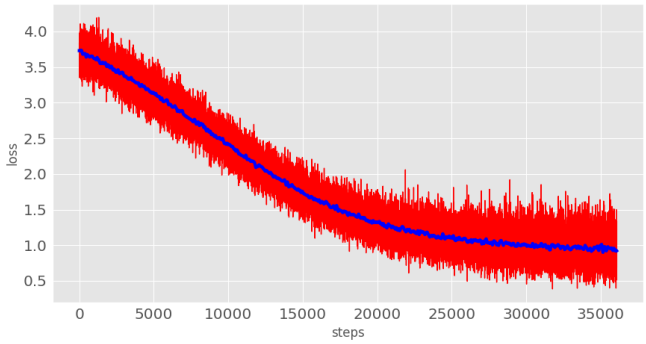
Visual

在训练过程中，观察损失函数和准确度的变化可以帮助优化超参数的设计。

loss

```
python3 visual/loss.py
```

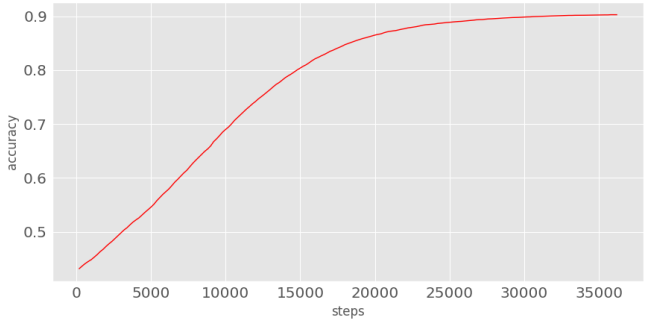
Figure 9: 损失函数曲线



accuracy

```
python3 visual/accuracy.py
```

Figure 10: 准确度曲线



在迭代了 25,000 步后就基本维持在一个固定值，学习停止了。

Baseline

使用 *deep_qa_1* 获得的 Baseline 数据为:

```
Epoch 25, total step 36400, accuracy 0.9031, cost 1.056221.
```

总结

Baseline model设计的非常简单，它展示了如何使用insuranceqa-corpus-zh训练 FAQ 问答模型。在过去两周中，为了能让这个数据集能满足使用，体现其价值，我花了很多时间来建设，仓促之中仍然会包含一些不足，比如数据集中，每个问题

是唯一的，不包含相似问题，是这个数据集目前最大的缺陷，另外一方面，因为该数据集的回复包含一个正例和多个负例，可以用用于训练分类器，也可以用于训练 ranking model。如果在使用的过程中，遇到任何问题，可以通过数据集的地址[4]反馈。

[4]反馈地址, https://github.com/Samurais/insuranceqa-corpus-zh/issues