# PyTorch自然语言处理实战

1. PyTorch深度学习框架简介
2. 使用torchtext进行文本的预处理
3. 使用PyTorch构建自然语言处理模型
4. 使用GPU对模型进行训练
5. 使用混合前端部署模型
6. 总结

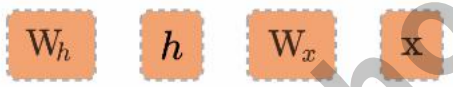
# PyTorch深度学习框架简介

- PyTorch是一个基于动态图的深度学习框架，支持动态构建神经网络，自动求导，分布式训练等特性。
- 使用Python语法灵活动态的构建神经网络，同时支持CPU/GPU上的计算，计算效率高
- 支持模块化搭建深度学习模型，方便复杂模型的构建

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```



- PyTorch的使用方法简单示例
- 使用torch.nn.Module来搭建神经网络模块

```
import torch
import torch.nn as nn
class Model(nn.Module):
    def __init__(self):
        super(Model, self).__init__()
        ...

    def forward(self, input):
        ...
```

# PyTorch构建自然语言处理模型的步骤

- 获取文本语料库 （通过爬虫等方法收集训练数据）
- 文本数据的清洗 （删除无用数据，冗余数据和乱码等等）
- 文本预处理 （正则化，分词，去停词），构建词库（给单词赋予序号）
- 文本转化成对应序号，输入自然语言处理模型进行训练
- 模型的验证，部署等后续步骤

# 使用torchtext进行文本的预处理

- torchtext: 适合于PyTorch的自然语言处理工具库，内置多种数据预处理工具和公开数据集
- torchtext.data包的使用：
  torchtext.data.Dataset构建数据集：支持csv, tsv, json等格式
  torchtext.data.Field构建数据集的一栏：支持不同分词方法，词库的构建，特殊单词（<sos>, <eos>, <unk>, <pad>等等）
  torchtext.dataset内置的特殊类型的数据集的支持，如机器翻译数据集等等
  torchtext.data.Iterator从数据集导出迭代器，输出最大句子长度x批尺寸的张量
- torchtext.vocab是torchtext.data.Field的成员变量，定义了词库，一般来说，四个特殊的词[‘<unk>’, ‘<pad>’, ‘<sos>’, ‘<eos>’]分别定义为0，1，2，3号单词

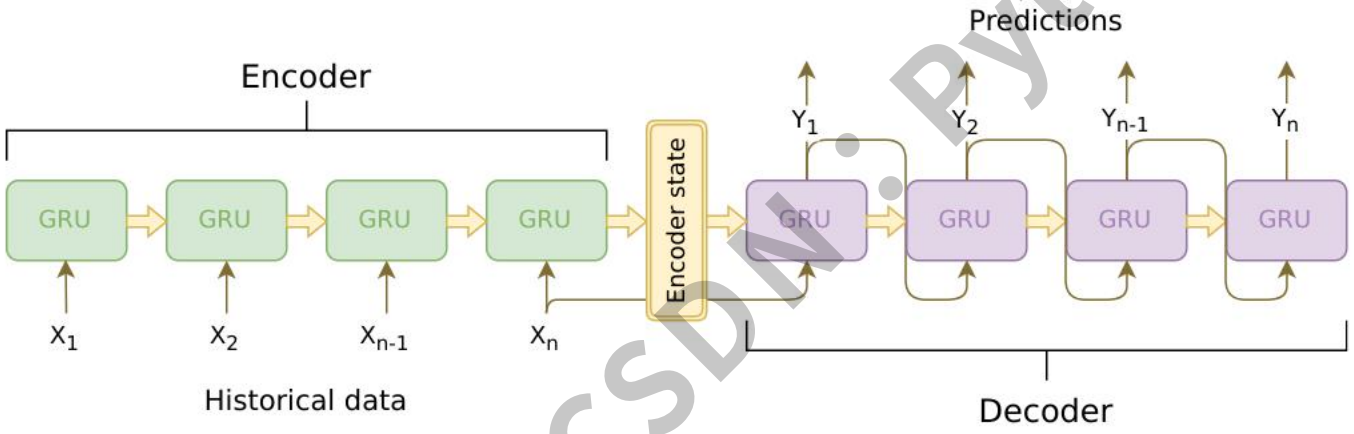
# 文本预处理实例代码

```
from torchtext import data, datasets
class DataLoader(object):
    def __init__(self, src_file, tgt_file):
        field = data.Field(init_token="<sos>", eos_token="<eos>")
        self.data = datasets.TranslationDataset("",
            exts=[src_file, tgt_file], fields=[field, field])
        field.build_vocab(self.data)
        self.field = field

    def get_batch_iter(self, bs=32):
        train_iter = data.BucketIterator(dataset=self.data,
            batch_size=bs,
            sort_key=lambda x: data.interleave_keys(len(x.src), len(x.trg)))
        return train_iter
```

# 使用PyTorch构建自然语言处理模型

- 使用nn.Embedding构建词向量 (词汇数目x词向量大小)
- 使用nn.GRU进行编码/解码
- 注意力机制和Seq2Seq模型的构建
- 损失函数和训练

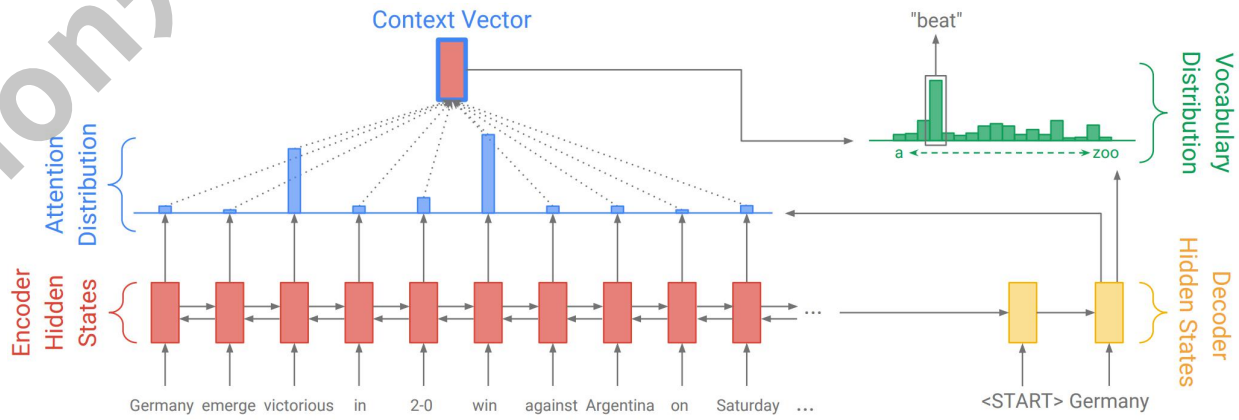
# Seq2Seq代码示例



```python
class Seq2Seq(nn.Module):

    def __init__(self, nvoc, nembed, nlayers=1, dropout=0):
        super(Seq2Seq, self).__init__()
        self.embed = nn.Embedding(nvoc, nembed) # 词嵌入模块
        self.enc = Encoder(nembed, nembed, nlayers, dropout) # 编码器
        self.attn = BahdanauAttn(2*nembed) # 注意力机制
        # 解码器
        self.dec = Decoder(nembed, nvoc, 2*nembed, self.attn, nlayers, dropout)
        self.nlayers = nlayers
        self.nhidden = nembed

    def forward(self, input, output):
        if self.training:
            input_embed = self.embed(input) # 根据单词id获取源语句词向量
            memory, state = self.enc(input_embed) # 词向量输入RNN
            enc_mask = (input == 1) # 获取词向量掩码
            state = state.view(self.nlayers, 2, -1, self.nhidden)
            state = state.transpose(1, 2).contiguous()\
                .view(self.nlayers, -1, 2*self.nhidden)
            scores = []
            # 解码过程，使用解码器获取每个单词概率
            for idx in range(output.shape[0]):
                output_ids = torch.unsqueeze(output[idx, :], 0)
                output_embed = self.embed(output_ids)
                score, state, _ = self.dec(memory, state, enc_mask, output_embed)
                scores.append(score)
            return torch.cat(scores, dim=0)
        else:
            ...
```



# 编码器代码示例

- 在PyTorch中，词汇id张量的格式一般是：最大长度x批尺寸
- 通过nn.Embedding后，词向量的格式一般是：最大长度x批尺寸x词向量大小
- 使用双向GRU模块，输入词向量
- nn.GRU模块输出两个结果，GRU的输出（最大长度x批尺寸x2*隐藏层大小）以及隐藏状态（2x批尺寸x隐藏层大小）

```
class Encoder(nn.Module):

    def __init__(self, nembed, nhidden, nlayers=1, dropout=0):
        super(Encoder, self).__init__()
        # 双向GRU，输入大小为词向量大小，输出大小为隐藏层大小
        self.gru = nn.GRU(nembed, nhidden, nlayers,
            dropout=dropout, bidirectional=True)

    def forward(self, input):
        return self.gru(input)
```

# 解码器/注意力机制代码示例

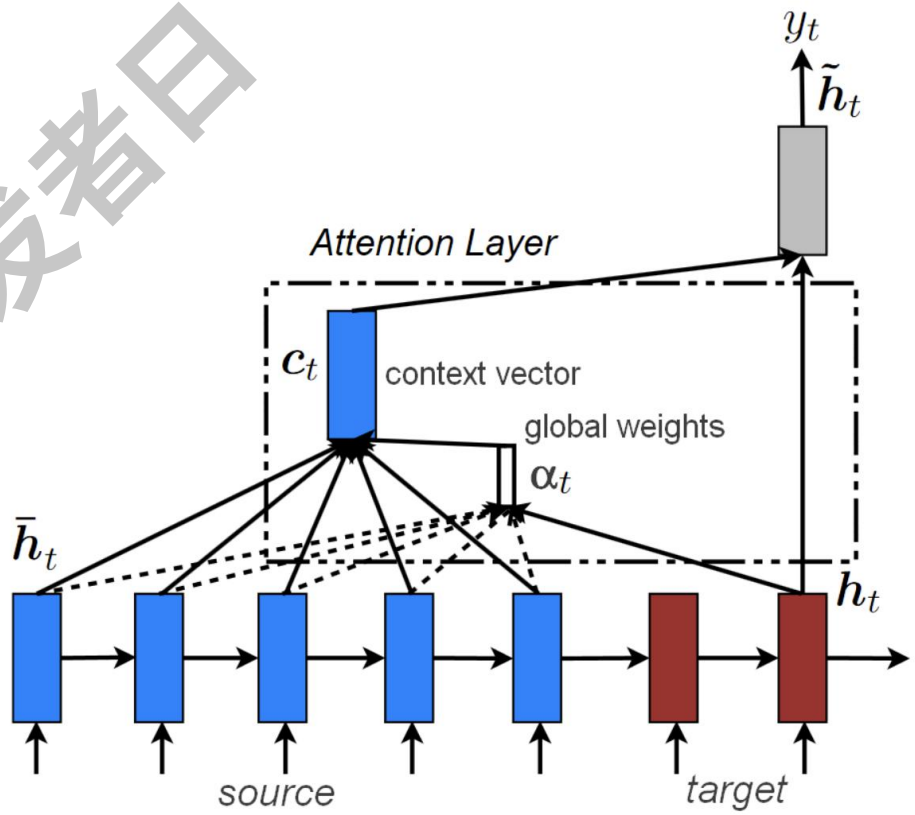


```python
class BahdanauAttn(nn.Module):
    def __init__(self, nhidden):
        super(BahdanauAttn, self).__init__()
        self.fc1 = nn.Linear(2*nhidden, nhidden)
        self.fc2 = nn.Linear(nhidden, 1, bias=False)

    def forward(self, memory, state, mask):
        ms = torch.cat((memory, state.expand_as(memory)), dim=-1)
        score = self.fc2(torch.tanh(self.fc1(ms)))
        score[mask] = -1e9 # 设置<pad>权重为0
        prob = torch.softmax(score, dim=0) # 计算源语句输出的权重
        return (prob*memory).sum(0, keepdim=True), prob.squeeze()

class Decoder(nn.Module):
    def __init__(self, nembed, nvoc, nhidden, attn, nlayers=1, dropout=0):
        super(Decoder, self).__init__()
        self.attn = attn
        self.gru = nn.GRU(nembed+nhidden, nhidden, nlayers,
            dropout=dropout, bidirectional=False)
        self.fc = nn.Linear(2*nhidden, nvoc)

    def forward(self, memory, state, enc_mask, input):
        context, prob = self.attn(memory, state, enc_mask) # 计算源语句输出加权平均
        output, state = self.gru(torch.cat((input, context), dim=-1), state) # 解码
        score = self.fc(torch.cat((output, context), dim=-1)) # 输出每个单词概率
        attn = prob
        return score, state, attn
```

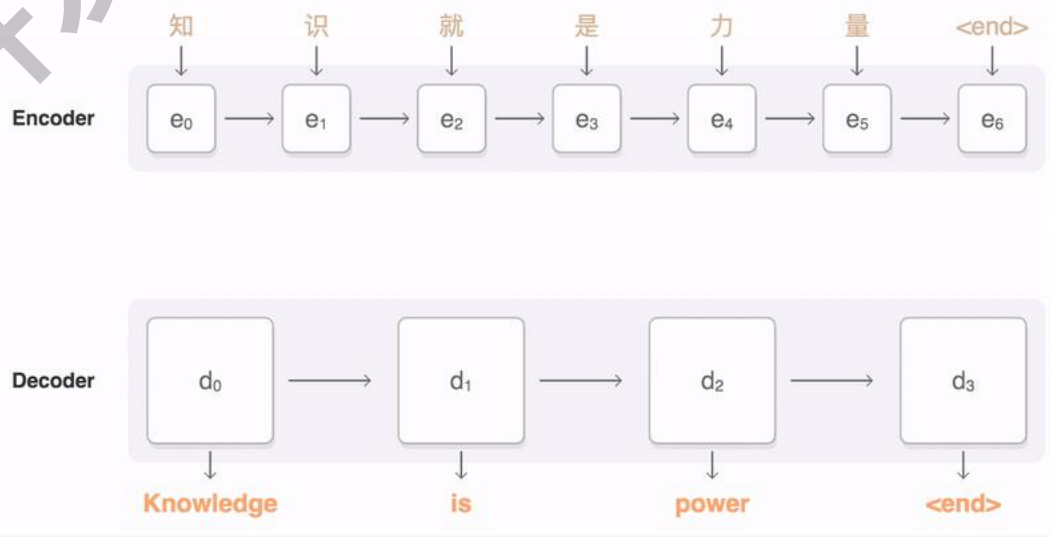



$$\text{score}(\boldsymbol{h}_t, \bar{\boldsymbol{h}}_s) = \begin{cases} \boldsymbol{h}_t^\top \bar{\boldsymbol{h}}_s & \textit{dot} \\ \boldsymbol{h}_t^\top \boldsymbol{W}_a \bar{\boldsymbol{h}}_s & \textit{general} \\ \boldsymbol{v}_a^\top \tanh\left(\boldsymbol{W}_a[\boldsymbol{h}_t; \bar{\boldsymbol{h}}_s]\right) & \textit{concat} \end{cases}$$

# 解码过程示例

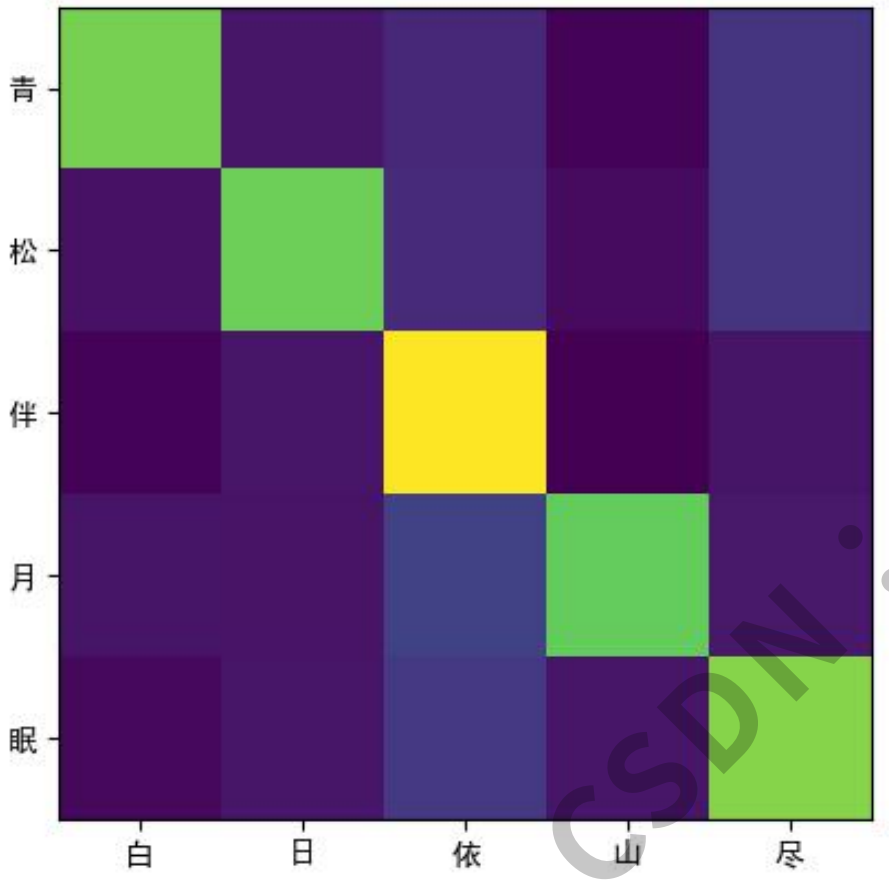


```
if self.training:
...
else:
    torch.set_grad_enabled(False) # 推断时不需要梯度，关闭梯度
    input_embed = self.embed(input) # 获取词向量
    memory, state = self.enc(input_embed) # 词向量编码
    enc_mask = (input == 1) # 获取掩码
    state = state.view(self.nlayers, 2, -1, self.nhidden)
    state = state.transpose(1, 2).contiguous()\
        .view(self.nlayers, -1, 2*self.nhidden)
    attns = []
    current_idx = output
    ret_idx = []
    rets = 0.0
    for idx in range(input.shape[0]):
        output_ids =  torch.unsqueeze(current_idx, 0)
        output_embed = self.embed(output_ids)
        score, state, attn = self.dec(memory, state, enc_mask, output_embed) # 解码
        prob = torch.softmax(score, dim=-1).squeeze() # 获取每个单词可能概率
        _, current_idx = prob.max(-1) # 获取当前最可能单词（贪心算法）
        rets += torch.log(prob[current_idx]) # 获取当前单词概率对数
        ret_idx.append(current_idx)
        current_idx = torch.tensor([current_idx]).to(input.device)
        attns.append(attn)
    torch.set_grad_enabled(True)
    return ret_idx, torch.stack(attns, dim=0), rets
```



# 注意力的可视化

- 通过输出源单词对目标单词的权重，可以得到单词之间的相互关系



# 使用GPU对模型进行训练



```python
def train():
    NEPOCHS = 100
    dl = DataLoader("dataset/couplet/train/in.txt", "dataset/couplet/train/out.txt")
    seq2seq = Seq2Seq(len(dl.field.vocab), 1024).cuda() # 转移模型到GPU上
    optimizer = torch.optim.Adam(seq2seq.parameters(), lr=1e-3) # 定义优化器
    criterion = SoftmaxCE(len(dl.field.vocab)).cuda() # 定义损失函数
    avg_loss = AverageMeter("Average Loss:", ":6.3f")
    for i in range(NEPOCHS):
        print("In epoch #{:3d}".format(i+1))
        for idx, b in enumerate(dl.get_batch_iter(256)):
            src, tgt = b.src, b.trg
            src = src.cuda() # 转移源语句到GPU上
            tgt = tgt.cuda() # 转移目标语句到GPU上
            pred = seq2seq(src, tgt)
            loss = criterion(pred, tgt) # 计算损失函数
            avg_loss.update(loss.item())
            optimizer.zero_grad() # 清空梯度
            loss.backward() # 计算梯度
            optimizer.step() # 优化器迭代
            print("In batch {:4d}".format(idx+1), avg_loss, end="\r")
        torch.save({ # 保存模型
            "model": seq2seq.state_dict(),
            "vocab": dl.field.vocab
            },
            "./model_epoch{:03d}.pt".format(i+1))
        print("")
```

# 使用混合前端部署模型



```python
class Seq2Seq(torch.jit.ScriptModule):

    def __init__(self, nvoc, nembed, nlayers=1, dropout=0):
        super(Seq2Seq, self).__init__()
        self.embed = nn.Embedding(nvoc, nembed)
        max_len = 10
        # 随机产生输入参数， 用于即时编译器（JIT）编译成静态图
        fake1 = torch.randn(max_len, 1, nembed)
        fake2 = torch.randn(max_len, 1, 2*nembed)
        fake3 = torch.randn(1, 1, 2*nlayers*nembed)
        fake4 = torch.randint(0, 2, (max_len, 1)).to(torch.uint8)
        fake5 = torch.randn(1, 1, nembed)

        enc = Encoder(nembed, nembed, nlayers, dropout)
        self.enc = torch.jit.trace(enc, (fake1, ))
        attn = BahdanauAttn(2*nembed)
        self.attn = torch.jit.trace(attn, (fake2, fake3, fake4))
        dec = Decoder(nembed, nvoc, 2*nembed, self.attn, nlayers, dropout)
        self.dec = torch.jit.trace(dec, (fake2, fake3, fake4, fake5))
        self.nlayers = nlayers
        self.nhidden = nembed
        self.max_len = max_len

    __constants__ = ['nlayers', 'nhidden', 'max_len']

    @torch.jit.script_method # JIT编译方法为静态图
    def fowrard(self, input):
        ...
        return output
```

# 总结

- 使用PyTorch深度学习框架来快速构建深度学习模型
- 使用torchtext来加载语料文本
- 模型的训练和部署

CSDN

2019
Python开发者日

让开发者紧跟
技术潮流

# Q&A

CSDN

2019
Python开发者日

让开发者紧跟
技术潮流

感谢聆听！