变压器注解及PyTorch实现（下）

训练

-批和蒽

-训练循环

-训练数据和批处理

-硬件和训练进度

-优化器

-正则化

-标签平滑

第一个例子

-数据生成

-损失计算

-贪心解码

真实示例

-数据加载

-迭代器

-多GPU训练

-训练系统附加组件：BPE，搜索，平均

结果

-聚焦可视化

摘要

训练

本节介绍模型的训练方法。

> 快速穿插介绍训练标准编码器解码器模型需要的一些工具。首先我们定义一个包含源和目标句子的批训练对象进行训练，同时构造构造。

批和阳离子

```
class Batch:
    "Object for holding a batch of data with mask during training."
    def __init__(self, src, trg=None, pad=0):
        self.src = src
        self.src_mask = (src != pad).unsqueeze(-2)
        if trg is not None:
            self.trg = trg[:, :-1]
            self.trg_y = trg[:, 1:]
            self.trg_mask = \
                self.make_std_mask(self.trg, pad)
            self.ntokens = (self.trg_y != pad).data.sum()
    @staticmethod
    def make_std_mask(tgt, pad):
        "Create a mask to hide padding and future words."
        tgt_mask = (tgt != pad).unsqueeze(-2)
        tgt_mask = tgt_mask & Variable(
            subsequent_mask(tgt.size(-1)).type_as(tgt_mask.data))
        return tgt_mask
```

> 接下来，我们创建一个通用的训练和叠加函数来跟踪损失。我们放置一个通用的损失计算函数，它也处理参数更新。

训练循环

```
def run_epoch(data_iter, model, loss_compute):
    "Standard Training and Logging Function"
    start = time.time()
    total_tokens = 0
    total_loss = 0
    tokens = 0
    for i, batch in enumerate(data_iter):
        out = model.forward(batch.src, batch.trg,
                            batch.src_mask, batch.trg_mask)
        loss = loss_compute(out, batch.trg_y, batch.ntokens)
        total_loss += loss
        total_tokens += batch.ntokens
        tokens += batch.ntokens
        if i % 50 == 1:
            elapsed = time.time() - start
            print("Epoch Step: %d Loss: %f Tokens per Sec: %f" %
                    (i, loss / batch.ntokens, tokens / elapsed))
            start = time.time()
            tokens = 0
    return total_loss / total_tokens
```

训练数据和批处理

我们使用标准WMT 2014英语-德语数据集进行了训练，该数据集包含大约450万个句子对。 使用字节对的编码方法对句子进行编码，该编码具有大约37000个词的共享源-目标词汇表。 对于英语-法语，我们使用了WMT 2014 英语-法语数据集，该数据集由36M个句子组成，并将词分成32000个词片(Word-piece)的词汇表。

句子对按照近似的序列长度进行批处理。每个训练批包含一组句子对，包含大约25000个源词和25000个目标词。

> 我们将使用torch text来创建批次。下面更详细地讨论实现过程。 我们在torchtext的一个函数中创建批次，确保填充到最大批训练长度的大小不超过阈值（如果我们有8个GPU，则阈值为25000）。

```
global max_src_in_batch, max_tgt_in_batch
def batch_size_fn(new, count, sofar):
    "Keep augmenting batch and calculate total number of tokens + padding."
    global max_src_in_batch, max_tgt_in_batch
    if count == 1:
        max_src_in_batch = 0
        max_tgt_in_batch = 0
    max_src_in_batch = max(max_src_in_batch,  len(new.src))
    max_tgt_in_batch = max(max_tgt_in_batch,  len(new.trg) + 2)
    src_elements = count * max_src_in_batch
    tgt_elements = count * max_tgt_in_batch
    return max(src_elements, tgt_elements)
```

硬件和训练进度

我们在一台配备8个NVIDIA P100 GPU的机器上训练我们的模型。 对于使用本文所述的超参数的基本模型，每个训练单步大约需要0.4秒。 我们对基础模型进行了总共100,000步或12小时的训练。 对于我们的大型模型，每个训练单步时间为1.0秒。 大型模型通常需要训练300,000步（3.5天）。

优化器

我们选择Adam[1]作为优化器 ，其参数为 $\beta_1 = 0.9$ 、 $\beta_2 = 0.98$ 和 $\epsilon = 10^{-9}$ 。根据以下公式，我们在训练过程中改变了学习率：

$$\text{lrate} = \text{d}_{\text{model}}^{-0.5} \cdot \min(step\_num^{-0.5}, step\_num \cdot warmup\_steps^{-1.5})$$

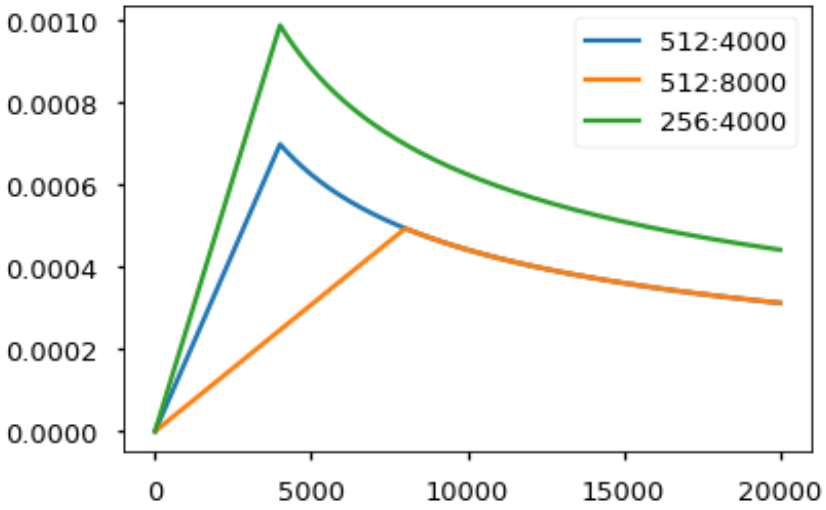
。在预热中随步数线性地增加学习速率，并且此后与步数的反平方根成比例地减小它。我们设置预热步数为4000。

> 注意：这部分非常重要，需要这种设置训练模型。

```
class NoamOpt:
    "Optim wrapper that implements rate."
    def __init__(self, model_size, factor, warmup, optimizer):
        self.optimizer = optimizer
        self._step = 0
        self.warmup = warmup
        self.factor = factor
        self.model_size = model_size
        self._rate = 0
    def step(self):
        "Update parameters and rate"
        self._step += 1
        rate = self.rate()
        for p in self.optimizer.param_groups:
            p['lr'] = rate
        self._rate = rate
        self.optimizer.step()
    def rate(self, step = None):
        "Implement `lrate` above"
        if step is None:
            step = self._step
        return self.factor * \
            (self.model_size ** (-0.5) *
            min(step ** (-0.5), step * self.warmup ** (-1.5)))
def get_std_opt(model):
    return NoamOpt(model.src_embed[0].d_model, 2, 4000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
```

> 当前模型在不同模型大小和超参数的情况下的曲线示例。

```
# Three settings of the lrate hyperparameters.
opts = [NoamOpt(512, 1, 4000, None),
        NoamOpt(512, 1, 8000, None),
        NoamOpt(256, 1, 4000, None)]
plt.plot(np.arange(1, 20000), [[opt.rate(i) for opt in opts] for i in range(1, 20000)])
plt.legend(["512:4000", "512:8000", "256:4000"])
None
```



正则化

标签平滑

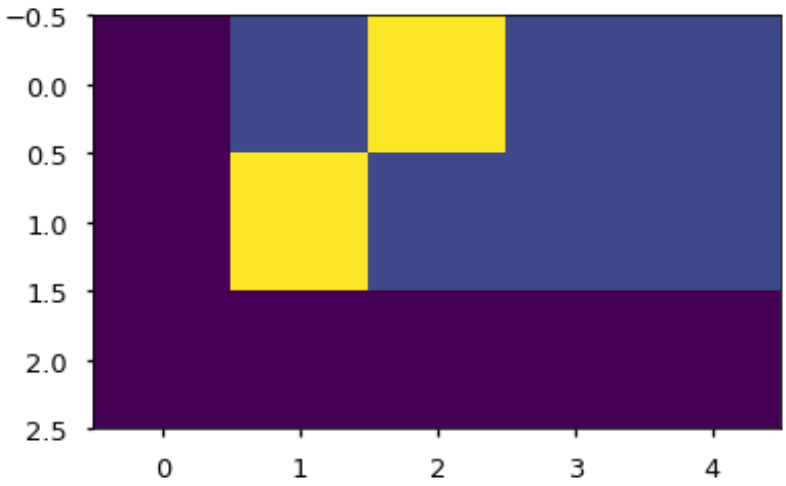
在训练期间，我们采用了值 $\epsilon_{ls} = 0.1$ [2]的标签平滑。 这种做法提高了困惑度，因为模型变得更加不确定，但提高了准确性和BLEU分数。

> 我们使用KL div loss实现标签平滑。 相比使用独热目标分布，我们创建一个分布，其包含正确单词的置信度和整个词汇表中分布的其余平滑项。

```
class LabelSmoothing(nn.Module):
    "Implement label smoothing."
    def __init__(self, size, padding_idx, smoothing=0.0):
        super(LabelSmoothing, self).__init__()
        self.criterion = nn.KLDivLoss(size_average=False)
        self.padding_idx = padding_idx
        self.confidence = 1.0 - smoothing
        self.smoothing = smoothing
        self.size = size
        self.true_dist = None
    def forward(self, x, target):
        assert x.size(1) == self.size
        true_dist = x.data.clone()
        true_dist.fill_(self.smoothing / (self.size - 2))
        true_dist.scatter_(1, target.data.unsqueeze(1), self.confidence)
        true_dist[:, self.padding_idx] = 0
        mask = torch.nonzero(target.data == self.padding_idx)
        if mask.dim() > 0:
            true_dist.index_fill_(0, mask.squeeze(), 0.0)
        self.true_dist = true_dist
        return self.criterion(x, Variable(true_dist, requires_grad=False))
```

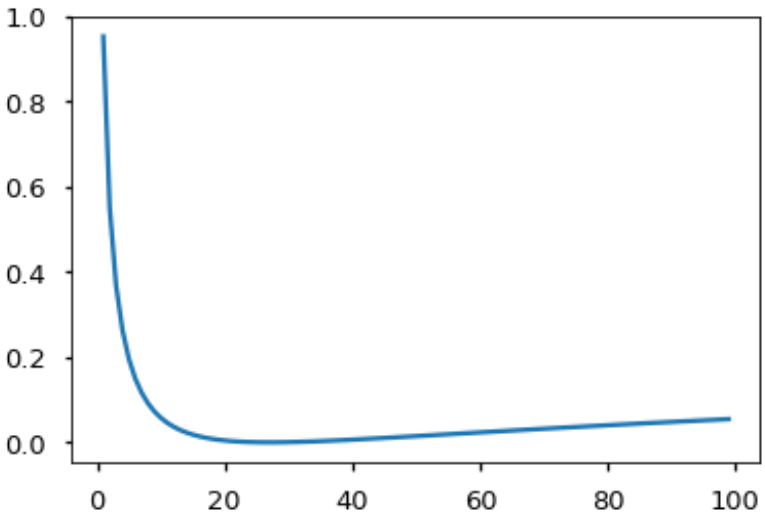
在这里，我们可以看到标签平滑的示例。

```
# Example of label smoothing.
crit = LabelSmoothing(5, 0, 0.4)
predict = torch.FloatTensor([[0, 0.2, 0.7, 0.1, 0],
                             [0, 0.2, 0.7, 0.1, 0],
                             [0, 0.2, 0.7, 0.1, 0]])
v = crit(Variable(predict.log()),
         Variable(torch.LongTensor([2, 1, 0])))
# Show the target distributions expected by the system.
plt.imshow(crit.true_dist)
None
```



> 如果对给定的选择非常有信心，标签平滑实际上会开始惩罚模型。

```
crit = LabelSmoothing(5, 0, 0.1)
def loss(x):
    d = x + 3 * 1
    predict = torch.FloatTensor([[0, x / d, 1 / d, 1 / d, 1 / d],
                                 ])
    #print(predict)
    return crit(Variable(predict.log()),
                Variable(torch.LongTensor([1]))).data[0]
plt.plot(np.arange(1, 100), [loss(x) for x in range(1, 100)])
None
```



第一个例子

> 我们可以先尝试一个简单的复制任务。 给定来自小词汇表的随机输入符号集，目标是生成那些相同的符号。

数据生成

```
def data_gen(V, batch, nbatches):
    "Generate random data for a src-tgt copy task."
    for i in range(nbatches):
        data = torch.from_numpy(np.random.randint(1, V, size=(batch, 10)))
        data[:, 0] = 1
        src = Variable(data, requires_grad=False)
        tgt = Variable(data, requires_grad=False)
        yield Batch(src, tgt, 0)
```

损失计算

```
class SimpleLossCompute:
    "A simple loss compute and train function."
    def __init__(self, generator, criterion, opt=None):
        self.generator = generator
        self.criterion = criterion
        self.opt = opt
    def __call__(self, x, y, norm):
        x = self.generator(x)
        loss = self.criterion(x.contiguous().view(-1, x.size(-1)),
                              y.contiguous().view(-1)) / norm
        loss.backward()
        if self.opt is not None:
            self.opt.step()
            self.opt.optimizer.zero_grad()
        return loss.data[0] * norm
```

贪心解码

```
# Train the simple copy task.
V = 11
criterion = LabelSmoothing(size=V, padding_idx=0, smoothing=0.0)
model = make_model(V, V, N=2)
model_opt = NoamOpt(model.src_embed[0].d_model, 1, 400,
        torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
for epoch in range(10):
    model.train()
    run_epoch(data_gen(V, 30, 20), model,
              SimpleLossCompute(model.generator, criterion, model_opt))
    model.eval()
    print(run_epoch(data_gen(V, 30, 5), model,
                    SimpleLossCompute(model.generator, criterion, None)))
```

```
Epoch Step: 1 Loss: 3.023465 Tokens per Sec: 403.074173
Epoch Step: 1 Loss: 1.920030 Tokens per Sec: 641.689380
1.9274832487106324
Epoch Step: 1 Loss: 1.940011 Tokens per Sec: 432.003378
Epoch Step: 1 Loss: 1.699767 Tokens per Sec: 641.979665
1.657595729827881
Epoch Step: 1 Loss: 1.860276 Tokens per Sec: 433.320240
Epoch Step: 1 Loss: 1.546011 Tokens per Sec: 640.537198
1.4888023376464843
Epoch Step: 1 Loss: 1.682198 Tokens per Sec: 432.092305
Epoch Step: 1 Loss: 1.313169 Tokens per Sec: 639.441857
1.3485562801361084
Epoch Step: 1 Loss: 1.278768 Tokens per Sec: 433.568756
Epoch Step: 1 Loss: 1.062384 Tokens per Sec: 642.542067
0.9853351473808288
```

16. Epoch Step: 1 Loss: 1.269471 Tokens per Sec: 433.388727
17. Epoch Step: 1 Loss: 0.590709 Tokens per Sec: 642.862135
18. 0.5686767101287842
19. Epoch Step: 1 Loss: 0.997076 Tokens per Sec: 433.009746
20. Epoch Step: 1 Loss: 0.343118 Tokens per Sec: 642.288427
21. 0.34273059368133546
22. Epoch Step: 1 Loss: 0.459483 Tokens per Sec: 434.594030
23. Epoch Step: 1 Loss: 0.290385 Tokens per Sec: 642.519464
24. 0.2612409472465515
25. Epoch Step: 1 Loss: 1.031042 Tokens per Sec: 434.557008
26. Epoch Step: 1 Loss: 0.437069 Tokens per Sec: 643.630322
27. 0.4323212027549744
28. Epoch Step: 1 Loss: 0.617165 Tokens per Sec: 436.652626
29. Epoch Step: 1 Loss: 0.258793 Tokens per Sec: 644.372296
30. 0.27331129014492034

> 为简单起见，此代码使用贪心解码来预测翻译。

```
def greedy_decode(model, src, src_mask, max_len, start_symbol):
    memory = model.encode(src, src_mask)
    ys = torch.ones(1, 1).fill_(start_symbol).type_as(src.data)
    for i in range(max_len-1):
        out = model.decode(memory, src_mask,
                           Variable(ys),
                           Variable(subsequent_mask(ys.size(1))
                                    .type_as(src.data)))
        prob = model.generator(out[:, -1])
        _, next_word = torch.max(prob, dim = 1)
        next_word = next_word.data[0]
        ys = torch.cat([ys,
                        torch.ones(1, 1).type_as(src.data).fill_(next_word)], dim=1)
    return ys
model.eval()
src = Variable(torch.LongTensor([[1,2,3,4,5,6,7,8,9,10]]) )
src_mask = Variable(torch.ones(1, 1, 10) )
```

```
print(greedy_decode(model, src, src_mask, max_len=10, start_symbol=1))
   1    2    3    4    5    6    7    8    9   10
[torch.LongTensor of size 1x10]
```

真实示例

> 现在我们通过IWSLT德语-英语翻译任务介绍一个真实示例。 该任务比上文提及的WMT任务小得多，但它说明了整个系统。 我们还展示了如何使用多个GPU处理加速其训练。

```
#!pip install torchtext spacy
#!python -m spacy download en
#!python -m spacy download de
```

数据加载

> 我们将使用torchtext和spacy加载数据集以进行词语切分。

```
# For data loading.
from torchtext import data, datasets
if True:
    import spacy
    spacy_de = spacy.load('de')
    spacy_en = spacy.load('en')
    def tokenize_de(text):
        return [tok.text for tok in spacy_de.tokenizer(text)]
    def tokenize_en(text):
        return [tok.text for tok in spacy_en.tokenizer(text)]
    BOS_WORD = '<s>'
    EOS_WORD = '</s>'
    BLANK_WORD = "<blank>"
    SRC = data.Field(tokenize=tokenize_de, pad_token=BLANK_WORD)
    TGT = data.Field(tokenize=tokenize_en, init_token = BOS_WORD,
                     eos_token = EOS_WORD, pad_token=BLANK_WORD)
    MAX_LEN = 100
    train, val, test = datasets.IWSLT.splits(
        exts=('.de', '.en'), fields=(SRC, TGT),
        filter_pred=lambda x: len(vars(x)['src']) <= MAX_LEN and
            len(vars(x)['trg']) <= MAX_LEN)
    MIN_FREQ = 2
    SRC.build_vocab(train.src, min_freq=MIN_FREQ)
    TGT.build_vocab(train.trg, min_freq=MIN_FREQ)
```

> 批训练对于速度来说很重要。我们希望批次分割非常均匀并且填充最少。 要做到这一点，我们必须修改torchtext默认的批处理函数。 这部分代码修补其默认批处理函数，以确保我们搜索足够多的句子以构建紧密批处理。

迭代器

```
class MyIterator(data.Iterator):
    def create_batches(self):
        if self.train:
            def pool(d, random_shuffler):
                for p in data.batch(d, self.batch_size * 100):
                    p_batch = data.batch(
                        sorted(p, key=self.sort_key),
                        self.batch_size, self.batch_size_fn)
                    for b in random_shuffler(list(p_batch)):
                        yield b
            self.batches = pool(self.data(), self.random_shuffler)
        else:
            self.batches = []
            for b in data.batch(self.data(), self.batch_size,
                                      self.batch_size_fn):
                self.batches.append(sorted(b, key=self.sort_key))
def rebatch(pad_idx, batch):
    "Fix order in torchtext to match ours"
    src, trg = batch.src.transpose(0, 1), batch.trg.transpose(0, 1)
    return Batch(src, trg, pad_idx)
```

多GPU训练

> 最后为了真正地快速训练，我们将使用多个GPU。 这部分代码实现了多GPU字生成。 它不是Transformer特有的，所以我不会详细介绍。 其思想是将训练时的单词生成分成块，以便在许多不同的GPU上并行处理。 我们使用PyTorch并行原语来做到这一点：

- 复制 - 将模块拆分到不同的GPU上
- 分散 - 将批次拆分到不同的GPU上

- 并行应用 - 在不同GPU上将模块应用于批处理
- 聚集 - 将分散的数据聚集到一个GPU上
- nn.DataParallel - 一个特殊的模块包装器，在评估之前调用它们。

```
# Skip if not interested in multigpu.
class MultiGPULossCompute:
    "A multi-gpu loss compute and train function."
    def __init__(self, generator, criterion, devices, opt=None, chunk_size=5):
        # Send out to different gpus.
        self.generator = generator
        self.criterion = nn.parallel.replicate(criterion,
                                               devices=devices)
        self.opt = opt
        self.devices = devices
        self.chunk_size = chunk_size
    def __call__(self, out, targets, normalize):
        total = 0.0
        generator = nn.parallel.replicate(self.generator,
                                          devices=self.devices)
        out_scatter = nn.parallel.scatter(out,
                                          target_gpus=self.devices)
        out_grad = [[] for _ in out_scatter]
        targets = nn.parallel.scatter(targets,
                                      target_gpus=self.devices)
        # Divide generating into chunks.
        chunk_size = self.chunk_size
        for i in range(0, out_scatter[0].size(1), chunk_size):
            # Predict distributions
            out_column = [[Variable(o[:, i:i+chunk_size].data,
                                    requires_grad=self.opt is not None)]
```

```
                    for o in out_scatter]
        gen = nn.parallel.parallel_apply(generator, out_column)
        # Compute loss.
        y = [(g.contiguous().view(-1, g.size(-1)),
              t[:, i:i+chunk_size].contiguous().view(-1))
             for g, t in zip(gen, targets)]
        loss = nn.parallel.parallel_apply(self.criterion, y)
        # Sum and normalize loss
        l = nn.parallel.gather(loss,
                               target_device=self.devices[0])
        l = l.sum()[0] / normalize
        total += l.data[0]
        # Backprop loss to output of transformer
        if self.opt is not None:
            l.backward()
            for j, l in enumerate(loss):
                out_grad[j].append(out_column[j][0].grad.data.clone())
    # Backprop all loss through transformer.
    if self.opt is not None:
        out_grad = [Variable(torch.cat(og, dim=1)) for og in out_grad]
        o1 = out
        o2 = nn.parallel.gather(out_grad,
                                target_device=self.devices[0])
        o1.backward(gradient=o2)
        self.opt.step()
        self.opt.optimizer.zero_grad()
    return total * normalize
```

> 现在我们创建模型，损失函数，优化器，数据迭代器和并行化。

```
# GPUs to use
devices = [0, 1, 2, 3]
if True:
    pad_idx = TGT.vocab.stoi["<blank>"]
    model = make_model(len(SRC.vocab), len(TGT.vocab), N=6)
    model.cuda()
    criterion = LabelSmoothing(size=len(TGT.vocab), padding_idx=pad_idx,
smoothing=0.1)
    criterion.cuda()
    BATCH_SIZE = 12000
    train_iter = MyIterator(train, batch_size=BATCH_SIZE, device=0,
                            repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                            batch_size_fn=batch_size_fn, train=True)
    valid_iter = MyIterator(val, batch_size=BATCH_SIZE, device=0,
                            repeat=False, sort_key=lambda x: (len(x.src), len(x.trg)),
                            batch_size_fn=batch_size_fn, train=False)
    model_par = nn.DataParallel(model, device_ids=devices)
None
```

> 现在我们训练模型。 我将稍微使用预热步骤，但其他一切都使用默认参数。 在具有4个Tesla V100 GPU的AWS p3.8xlarge机器上，每秒运行约27,000个词，批训练大小大小为12,000。

训练系统

```
#!wget https://s3.amazonaws.com/opennmt-models/iwslt.pt
```

```
if False:
    model_opt = NoamOpt(model.src_embed[0].d_model, 1, 2000,
            torch.optim.Adam(model.parameters(), lr=0, betas=(0.9, 0.98), eps=1e-9))
    for epoch in range(10):
        model_par.train()
        run_epoch((rebatch(pad_idx, b) for b in train_iter),
                model_par,
                MultiGPULossCompute(model.generator, criterion,
                                    devices=devices, opt=model_opt))
        model_par.eval()
        loss = run_epoch((rebatch(pad_idx, b) for b in valid_iter),
                         model_par,
                         MultiGPULossCompute(model.generator, criterion,
                         devices=devices, opt=None))
        print(loss)
else:
    model = torch.load("iwslt.pt")
```

> 一旦训练完成，我们可以解码模型以产生一组翻译。 在这里，我们只需翻译验证集中的第一个句子。 此数据集非常小，因此使用贪婪搜索的翻译相当准确。

```
for i, batch in enumerate(valid_iter):
    src = batch.src.transpose(0, 1)[:1]
    src_mask = (src != SRC.vocab.stoi["<blank>"]).unsqueeze(-2)
    out = greedy_decode(model, src, src_mask,
                        max_len=60, start_symbol=TGT.vocab.stoi["<s>"])
    print("Translation:", end="\t")
    for i in range(1, out.size(1)):
        sym = TGT.vocab.itos[out[0, i]]
        if sym == "</s>": break
        print(sym, end =" ")
    print()
    print("Target:", end="\t")
    for i in range(1, batch.trg.size(0)):
        sym = TGT.vocab.itos[batch.trg.data[i, 0]]
        if sym == "</s>": break
        print(sym, end =" ")
    print()
    break
```

```
Translation:    <unk> <unk> . In my language , that means , thank you very much .
Gold:    <unk> <unk> . It means in my language , thank you very much .
```

附加组件：BPE，搜索，平均

> 所以这主要涵盖了Transformer模型本身。 有四个方面我们没有明确涵盖。 我们还实现了所有这些附加功能 OpenNMT-py[3].
>
> 1) 字节对编码/ 字片(Word-piece)：我们可以使用库来首先将数据预处理为子字单元。参见Rico Sennrich的subword-nmt实现[4]。这些模型将训练数据转换为如下所示：

_Die _Protokoll datei _kann _ heimlich _per _E - Mail _oder _FTP _an _einen _bestimmte n _Empfänger _gesendet _werden .

> 2) 共享嵌入：当使用具有共享词汇表的BPE时，我们可以在源/目标/生成器之间共享相同的权重向量，详细见[5]。 要将其添加到模型，只需执行以下操作：

```
if False:
    model.src_embed[0].lut.weight = model.tgt_embeddings[0].lut.weight
    model.generator.lut.weight = model.tgt_embed[0].lut.weight
```

> 3) 集束搜索：这里展开说有点太复杂了。 PyTorch版本的实现可以参考 OpenNMT-py[6]。
>
> 4) 模型平均：这篇文章平均最后k个检查点以创建一个集合效果。 如果我们有一堆模型，我们可以在事后这样做：

```
def average(model, models):
    "Average models into model"
    for ps in zip(*[m.params() for m in [model] + models]):
        p[0].copy_(torch.sum(*ps[1:]) / len(ps[1:]))
```

结果

在WMT 2014英语-德语翻译任务中，大型Transformer模型（表2中的Transformer（大））优于先前报告的最佳模型（包括集成的模型）超过2.0 BLEU，建立了一个新的最先进BLEU得分为28.4。 该模型的配置列于表3的底部。在8个P100 GPU的机器上，训练需要需要3.5天。 甚至我们的基础模型也超过了之前发布的所有模型和集成，而且只占培训成本的一小部分。

在WMT 2014英语-法语翻译任务中，我们的大型模型获得了41.0的BLEU分数，优于以前发布的所有单一模型，不到以前最先进技术培训成本的1/4 模型。 使用英语到法语训练的Transformer（大）模型使用dropout概率 P_{drop} = 0.1，而不是0.3。

```
Image(filename="images/results.png")
```

Table 2: The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost.

Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	**$3.3 \cdot 10^{18}$**	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

> 我们在这里编写的代码是基本模型的一个版本。 这里有系统完整训练的版本 (Example Models[7]).
>
> 通过上一节中的附加扩展，OpenNMT-py复制在EN-DE WMT上达到26.9。 在这里，我已将这些参数加载到我们的重新实现中。

```
!wget https://s3.amazonaws.com/opennmt-models/en-de-model.pt
```

```
model, SRC, TGT = torch.load("en-de-model.pt")
```

```
model.eval()
sent = "▁The ▁log ▁file ▁can ▁be ▁sent ▁secret ly ▁with ▁email ▁or ▁FTP ▁to ▁a ▁specified ▁receiver".split()
src = torch.LongTensor([[SRC.stoi[w] for w in sent]])
src = Variable(src)
src_mask = (src != SRC.stoi["<blank>"]).unsqueeze(-2)
out = greedy_decode(model, src, src_mask,
                    max_len=60, start_symbol=TGT.stoi["<s>"])
print("Translation:", end="\t")
trans = "<s> "
for i in range(1, out.size(1)):
    sym = TGT.itos[out[0, i]]
    if sym == "</s>": break
    trans += sym + " "
print(trans)
```

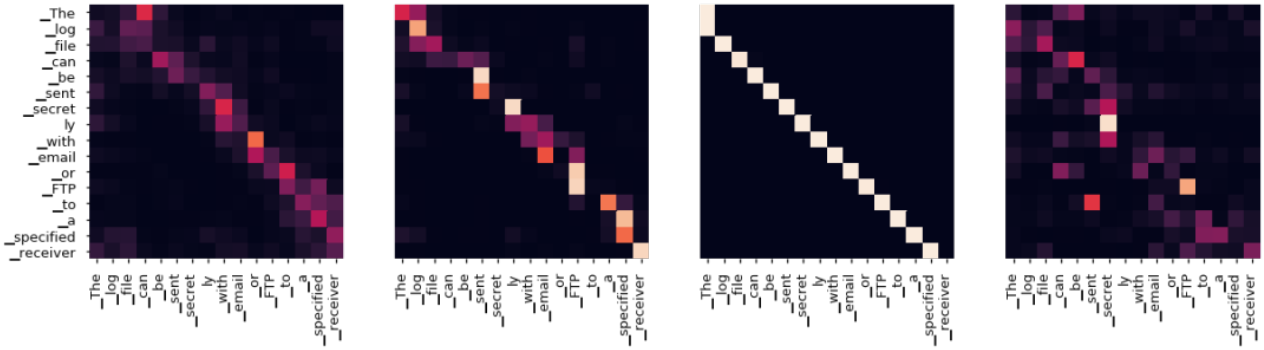
1. Translation: <s> _Die _Protokoll datei _kann _ heimlich _per _E - Mail _oder _FTP _an _einen _bestimmte n _Empfänger _gesendet _werden .

注意力可视化

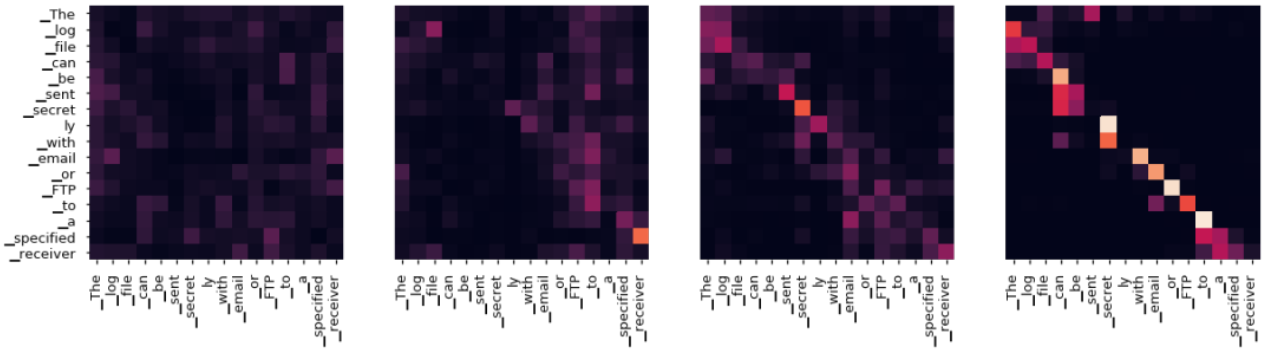
> 即使使用贪婪的解码器，翻译看起来也不错。 我们可以进一步想象它，看看每一层注意力发生了什么。

```
tgt_sent = trans.split()
def draw(data, x, y, ax):
    seaborn.heatmap(data,
                    xticklabels=x, square=True, yticklabels=y, vmin=0.0, vmax=1.0,
                    cbar=False, ax=ax)
for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Encoder Layer", layer+1)
    for h in range(4):
        draw(model.encoder.layers[layer].self_attn.attn[0, h].data,
            sent, sent if h ==0 else [], ax=axs[h])
    plt.show()
for layer in range(1, 6, 2):
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    print("Decoder Self Layer", layer+1)
    for h in range(4):
        draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(tgt_sent)],
            tgt_sent, tgt_sent if h ==0 else [], ax=axs[h])
    plt.show()
    print("Decoder Src Layer", layer+1)
    fig, axs = plt.subplots(1,4, figsize=(20, 10))
    for h in range(4):
        draw(model.decoder.layers[layer].self_attn.attn[0, h].data[:len(tgt_sent), :len(sent)],
            sent, tgt_sent if h ==0 else [], ax=axs[h])
    plt.show()
```

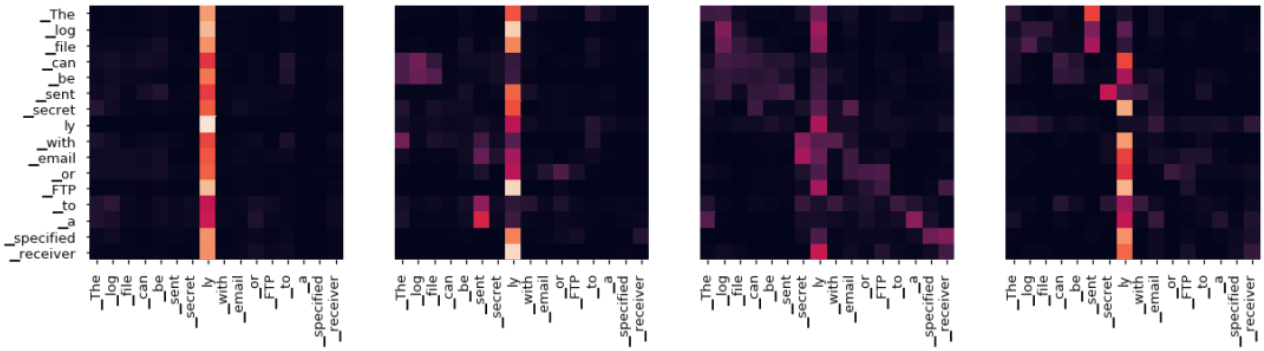
1. Encoder Layer 2



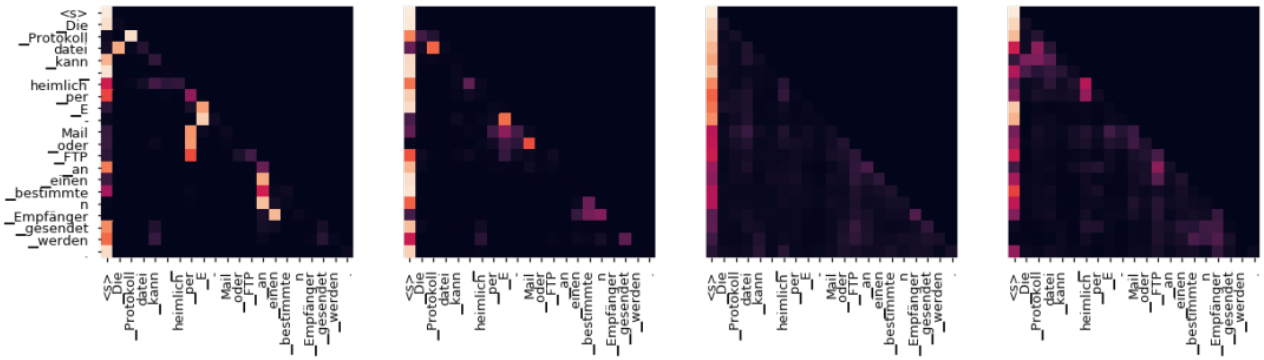
1. Encoder Layer 4



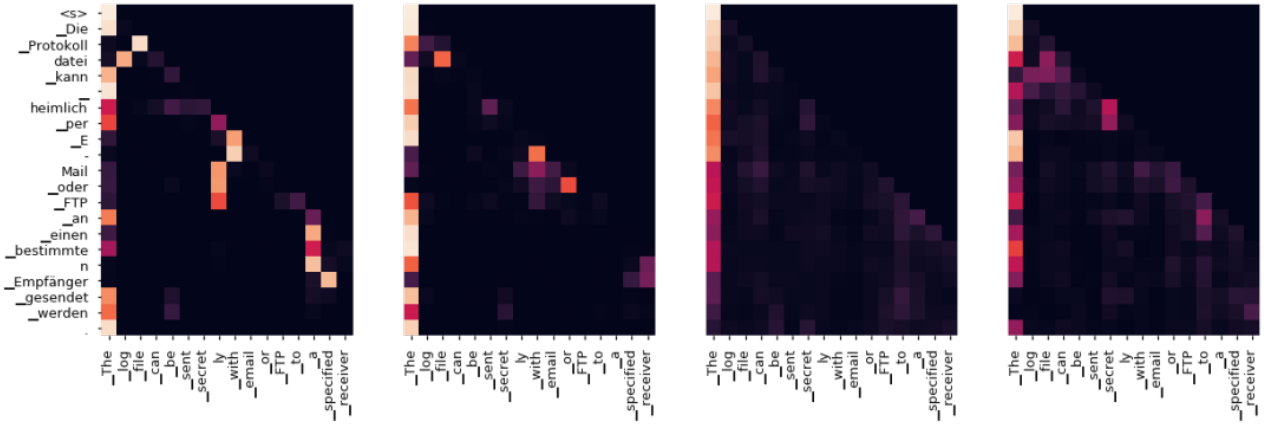
1. Encoder Layer 6



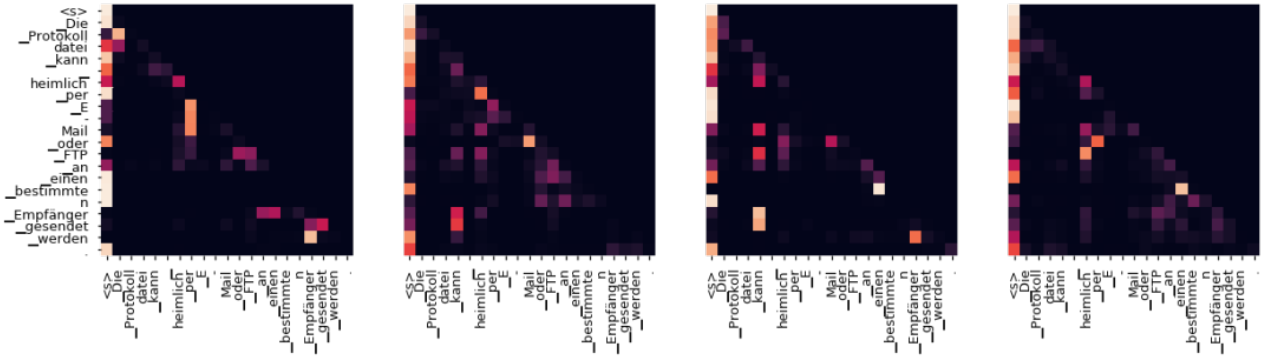
1. Decoder Self Layer 2



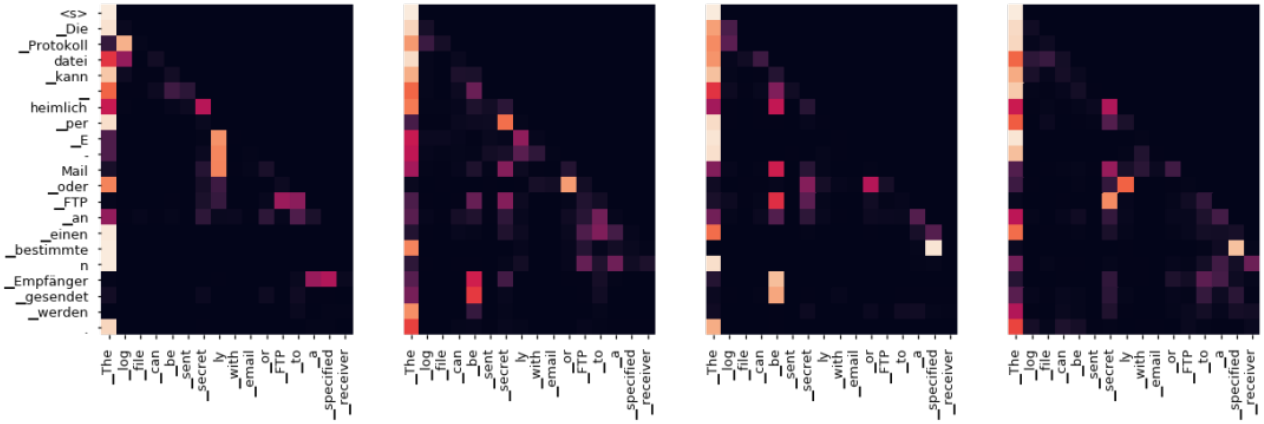
1. Decoder Src Layer 2



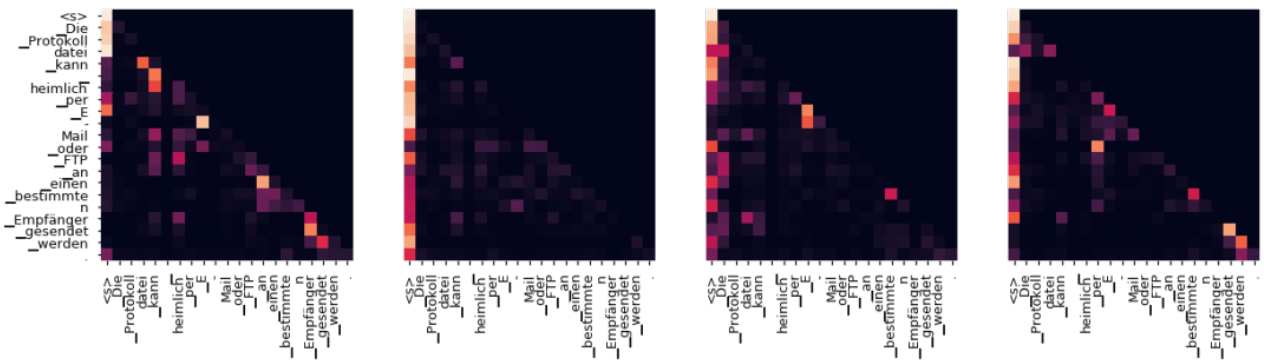
1. Decoder Self Layer 4



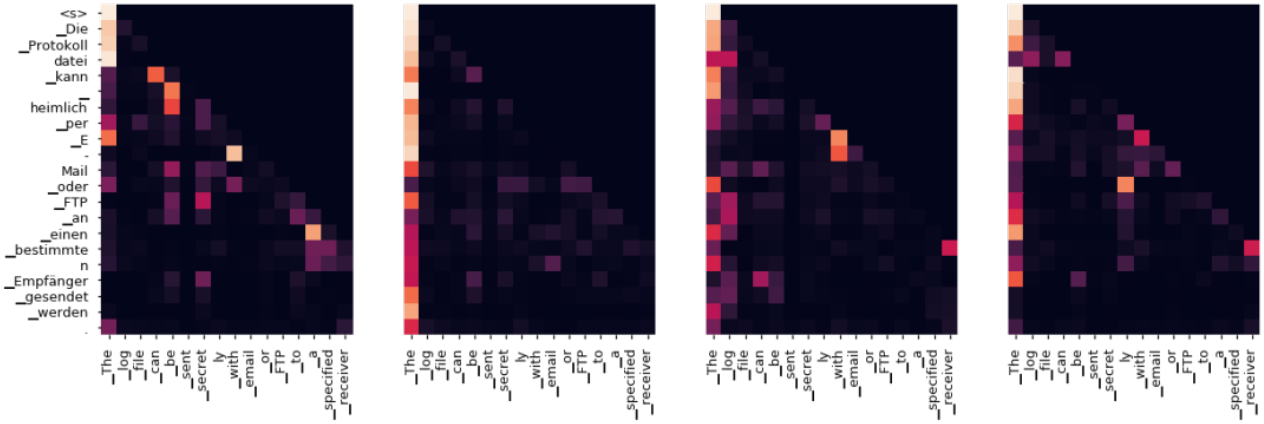
1. Decoder Src Layer 4



1. Decoder Self Layer 6



1. Decoder Src Layer 6



结论

> 希望这段代码对未来的研究很有用。 如果您有任何问题，请与我们联系。 如果您发现此代码有用，请查看我们的其他OpenNMT工具。

```
1.  @inproceedings{opennmt,
2.    author    = {Guillaume Klein and
3.                 Yoon Kim and
4.                 Yuntian Deng and
5.                 Jean Senellart and
6.                 Alexander M. Rush},
7.    title     = {OpenNMT: Open-Source Toolkit for Neural Machine Translation},
8.    booktitle = {Proc. ACL},
9.    year      = {2017},
10.   url       = {https://doi.org/10.18653/v1/P17-4012},
11.   doi       = {10.18653/v1/P17-4012}
12. }
```

> Cheers，srush

参考链接

[1] https://arxiv.org/abs/1412.6980

[2] https://arxiv.org/abs/1512.00567

[3] https://github.com/opennmt/opennmt-py

[4] https://github.com/rsennrich/subword-nmt

[5] https://arxiv.org/abs/1608.05859

[6] https://github.com/OpenNMT/OpenNMT-py/blob/master/onmt/translate/Beam.py

[7] http://opennmt.net/Models-py/

END