1. Machine Learning & Neural Networks

- (a) 关于Adam优化器（首次提出），PyTorch中的接口如下所示：

  ```
  torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08,
  weight_decay=0, amsgrad=False)
  ```

  `beta`参数的两个值的正如第(2)问中所示。

  - (1) 动量更新法则：保留当前点的信息（因为当前点的信息一定程度包含了之前所有更新迭代的信息，这有点类似LSTM与GRU的思想，但是此处并不会发生遗忘）

    $$\begin{aligned} m &\leftarrow \beta_1 m + (1-\beta_1)\nabla_\theta J_{\text{minibatch}}(\theta) \\ \theta &\leftarrow \theta - \alpha m \end{aligned} \tag{a3.1.1}$$

    注意β_1的取值默认为0.9，这表明会尽可能多地保留当前点的信息。

    从另一个角度来说，单纯的梯度下降法容易陷入局部最优，直观上来看，带动量的更新可以使得搜索路径呈现出一个弧形收敛的形状（有点像一个漩涡收敛到台风眼），因为每次更新不会偏离原先的方向太多，这样的策略容易跳出局部最优点，并且将搜索范围控制在一定区域内（漩涡内），容易最终收敛到全局最优。

  - (2) 完整的Adam优化器还使用了**自适应学习率**的技术：

    $$\begin{aligned} m &\leftarrow \beta_1 m + (1-\beta_1)\nabla_\theta J_{\text{minibatch}}(\theta) \\ v &\leftarrow \beta_2 v + (1-\beta_2)(\nabla_\theta J_{\text{minibatch}})(\theta) \odot \nabla_\theta J_{\text{minibatch}}(\theta)) \\ \theta &\leftarrow \theta - \alpha m / \sqrt{v} \end{aligned} \tag{a3.1.2}$$

    其中$\odot$与$/$运算符表示点对点的乘法与除法（上面的$\odot$相当于是梯度中所有元素取平方）。

    β_2默认值0.99，这里相当于做了学习率关于梯度值的自适应调整（每个参数的调整都不一样，注意$/$号是点对点的除法），在非稳态和在线问题上有很有优秀的性能。

    一般来说随着优化迭代，梯度值会逐渐变小（理想情况下最终收敛到零），因此v的取值应该会趋向于变小，步长则是变大，这个就有点奇怪了，理论上优化应该是前期大步长找到方向，后期小步长做微调。

    找到一篇详细总结Adam优化器优点的博客。

- (b) Dropout技术是在神经网络训练过程中以一定概率p_{drop}将隐层h中的若干值设为零，然后乘以一个常数γ，具体而言：

  $$h_{\text{drop}} = \gamma d \odot h \quad d \in \{0,1\}^n, h \in \mathbb{R}^n \tag{a3.1.3}$$

  这里之所以乘以γ是为了使得h中每个点位的期望值不变，即：

  $$\mathbb{E}_{p_{\text{drop}}}[h_{\text{drop}}]_i = h_i \tag{a3.1.4}$$

  - (1) 根据期望定义有如下推导：

    $$\mathbb{E}_{p_{\text{drop}}}[h_{\text{drop}}]_i = p_{\text{drop}} \cdot 0 + (1-p_{\text{drop}})\gamma h_i = h_i \Rightarrow \gamma = \frac{1}{1-p_{\text{drop}}} \tag{a3.1.5}$$

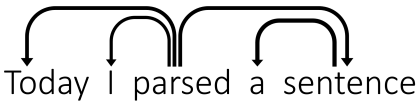
  - (2) Dropout是用来防止模型过拟合，缓解模型运算复杂度，评估的时候显然不能使用Dropout，因为用于评估的模型必须是确定的，Dropout是存在不确定性的。

2. Neural Transition-Based Dependency Parsing

本次使用的是PyTorch1.7.1CPU版本，当然使用GPU版本应该会更好。

本次实现的是基于Transition的依存分析模型，就是在实现[notes]中的**Greedy Deterministic Transition-Based Parsing**算法。其中**SHIFT**是将缓存中的第一个移入栈，**LEFT-ARC**与**RIGHT-ARC**分别是建立栈顶前两个单词之间的依存关系。

- (a) 具体每步迭代结果如下所示（默认ROOT是指向parsed的）：



Stack	Buffer	New dependency	Transition
[ROOT]	[Today, I, parsed, a, sentence]		Initial Configuration
[ROOT, Today]	[I, parsed, a, sentence]		SHIFT
[ROOT, Today, I]	[parsed, a, sentence]		SHIFT
[ROOT, Today, I, parsed]	[a, sentence]		SHIFT
[ROOT, Today, parsed]	[a, sentence]	parsed $\rightarrow$ I	LEFT-ARC
[ROOT, parsed]	[a, sentence]	parsed $\rightarrow$ Today	LEFT-ARC
[ROOT, parsed, a]	[sentence]		SHIFT
[ROOT, parsed, a, sentence]	[]		SHIFT
[ROOT, parsed, sentence]	[]	sentence $\rightarrow$ a	LEFT-ARC
[ROOT, parsed]	[]	parsed $\rightarrow$ sentence	RIGHT-ARC
[ROOT]	[]	ROOT $\rightarrow$ parsed	RIGHT-ARC

- (b) **SHIFT**共计n次，**LEFT-ARC**与**RIGHT-ARC**合计n次，共计$2n$次。
- (c) 非常简单的状态定义与转移定义代码实现，运行`python parser_transitions.py part_c`通过测试。
- (d) 运行`python parser_transitions.py part_d`通过测试。
- (e) 实现神经依存分析模型，参考的是**lecture4**推荐阅读的第二篇（A Fast and Accurate Dependency Parser using Neural Networks）。运行`python run.py`通过测试。

  注意这一题要求是自己实现全连接层和嵌入层的逻辑，不允许使用PyTorch内置的层接口，有兴趣的自己去实现吧，我就直接调用接口了。如果是要从头到尾都重写，这个显得就很困难（需要把反向传播和梯度计算的逻辑都要实现），然而本题的模型还是继承了`torch.nn.Module`的，因此似乎只能继承`torch.nn.Module`写自定义网络层，这样其实还是比较简单的，这可以参考我的博客2.1节的全连接层重写的代码。

  运行结果：

```
================================================================================
====
INITIALIZING
================================================================================
====
Loading data...
took 1.36 seconds
Building parser...
took 0.82 seconds
Loading pretrained embeddings...
took 2.48 seconds
Vectorizing data...
took 1.22 seconds
Preprocessing training data...
took 30.56 seconds
took 0.02 seconds

================================================================================
====
TRAINING
================================================================================
====
Epoch 1 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:18<00:00, 23.61it/s]
Average Train Loss: 0.18908768985420465
Evaluating on dev set
144585Oit [OO:OO, 46259788.38it/s]
- dev UAS: 83.75
New best dev UAS! Saving model.

Epoch 2 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:15<00:00, 24.52it/s]
Average Train Loss: 0.1157231591158099
Evaluating on dev set
144585Oit [OO:OO, 92527340.72it/s]
- dev UAS: 86.22
New best dev UAS! Saving model.

Epoch 3 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:14<00:00, 24.86it/s]
Average Train Loss: 0.1010169279418918
Evaluating on dev set
144585Oit [OO:OO, 61690227.55it/s]
- dev UAS: 87.04
New best dev UAS! Saving model.

Epoch 4 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:16<00:00, 24.17it/s]
Average Train Loss: 0.09254590892414381
```

```
Evaluating on dev set
144585Oit [OO:OO, 46221356.67it/s]
- dev UAS: 87.43
New best dev UAS! Saving model.

Epoch 5 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:16<00:00, 24.06it/s]
Average Train Loss: 0.08614181549977754
Evaluating on dev set
144585Oit [OO:OO, 46262964.5Oit/s]
- dev UAS: 87.72
New best dev UAS! Saving model.

Epoch 6 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:19<00:00, 23.2Oit/s]
Average Train Loss: 0.08176740852599859
Evaluating on dev set
144585Oit [OO:OO, 46264729.2Oit/s]
- dev UAS: 88.29
New best dev UAS! Saving model.

Epoch 7 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:17<00:00, 23.95it/s]
Average Train Loss: 0.07832196695343047
Evaluating on dev set
144585Oit [OO:OO, 45695793.4Oit/s]
- dev UAS: 88.17

Epoch 8 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
██| 1848/1848 [01:15<00:00, 24.4Oit/s]
Average Train Loss: 0.07501755065982153
Evaluating on dev set
144585Oit [OO:OO, 46264729.2Oit/s]
- dev UAS: 88.53
New best dev UAS! Saving model.

Epoch 9 out of 10
100%|
████████████████████████████████████████████████████████████████████████████
████████████████████████████████████████████████████████████████████████████|
1848/1848 [01:16<00:00, 24.15it/s]
Average Train Loss: 0.07205055564545192
Evaluating on dev set
144585Oit [OO:OO, 45701992.11it/s]
- dev UAS: 88.47

Epoch 10 out of 10
```

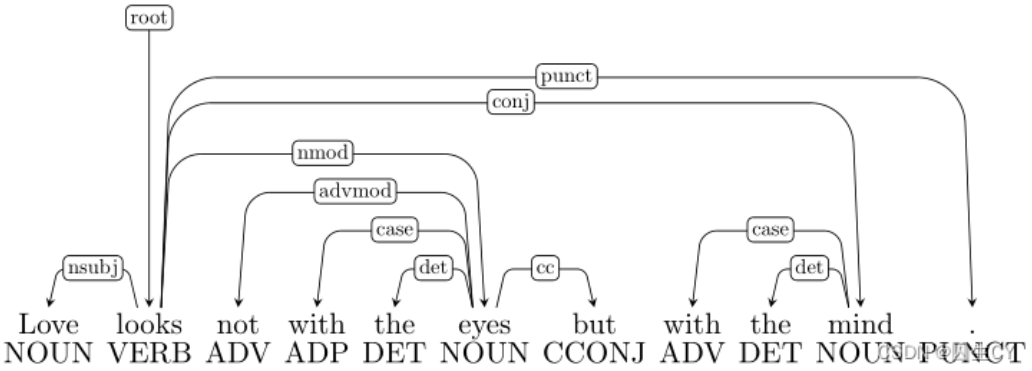
```
100%|
██████████████████████████████████████████████████████████████████████████████
██████████████████████████████████████████████████████████████████████████████|
1848/1848 [01:15<00:00, 24.54it/s]
Average Train Loss: 0.06958463928537258
Evaluating on dev set
1445850it [00:00, 46266141.05it/s]
- dev UAS: 88.76
New best dev UAS! Saving model.

================================================================================
====
TESTING
================================================================================
====
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 92289480.94it/s]
- test UAS: 89.15
Done!
```

作业中提到训练需要一个小时，使用GPU可以大大加快速度，训练过程中的损失函数值与UAS指数全部达标。（损失函数值应当低于0.2，UAS超过87%）

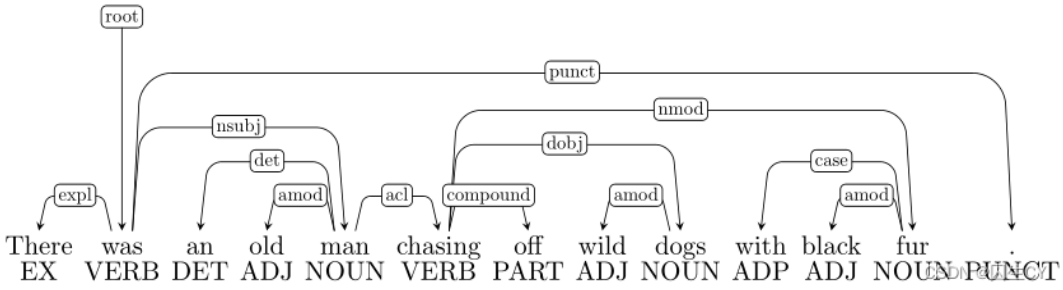
- (f) 这里提到几种解析错误类型：
  1. **介词短语依存错误**：sent into Afghanistan中正确的依存关系是sent → Afghanistan
  2. **动词短语依存错误**：Leaving the store unattended, I went outside to watch the parade中正确的依存关系是went指向leaving
  3. **修饰语依存错误**：I am extremely short中正确的依存关系是short → extremely
  4. **协同依存错误**：Would you like brown rice or garlic naan中短语brown rice和garlic naan是并列的，因此rice应当指向naan

  下面几小问不是那么确信，将就着看吧。

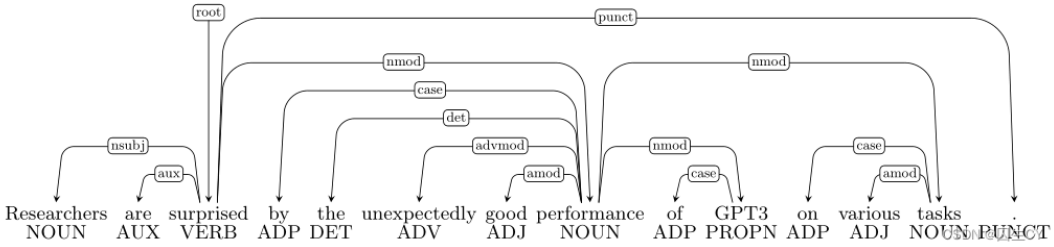
  - (1) 这个感觉是**介词短语依存错误**，但是looks的确指向eyes和mind了，这是符合上面的说法的。难道是**协同依存错误**?



  - (2) 这个感觉还是**介词短语依存错误**：chasing不该指向fur，fur应该是与dogs相互依存。



- (3) 这个很简单是unexpectedly和good之间属于**修饰语依存错误**，应当由good指向unexpectedly；



- (4) 这个根据排除法（没有介词短语，没有修饰词，也没有并列关系）只能是**动词短语依存错误**，但是具体是哪儿错了真的看不出来，可能是crossing和eating之间错标成了协同依存关系？