CS224n A3

유경

1. Machine Learning & Neural Networks (8 points)

(a) (4 points) Adam Optimizer

Recall the standard Stochastic Gradient Descent update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})$$

where $\boldsymbol{\theta}$ is a vector containing all of the model parameters, J is the loss function, $\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})$ is the gradient of the loss function with respect to the parameters on a minibatch of data, and α is the learning rate. Adam Optimization[1] uses a more sophisticated update rule with two additional steps.[2]

i. (2 points) First, Adam uses a trick called *momentum* by keeping track of $\mathbf{m}$, a rolling average of the gradients:

$$
\begin{aligned}
① \quad \mathbf{m} &\leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \mathbf{m}
\end{aligned}
$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain (you don't need to prove mathematically, just give an intuition) how using $\mathbf{m}$ stops the updates from varying as much and why this low variance may be helpful to learning, overall.

ii. (2 points) Adam also uses *adaptive learning rates* by keeping track of $\mathbf{v}$, a rolling average of the magnitudes of the gradients:

$$
\begin{aligned}
\mathbf{m} &\leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \\
② \quad \mathbf{v} &\leftarrow \beta_2 \mathbf{v} + (1 - \beta_2) (\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})) \\
\boldsymbol{\theta} &\leftarrow \boldsymbol{\theta} - \alpha \odot \mathbf{m} / \sqrt{\mathbf{v}}
\end{aligned}
$$

└ 제곱의효과 → 변동량(Variance)과 관련있음

where $\odot$ and $/$ denote elementwise multiplication and division (so $\mathbf{z} \odot \mathbf{z}$ is elementwise squaring) and β_2 is a hyperparameter between 0 and 1 (often set to 0.99). Since Adam divides the update by $\sqrt{\mathbf{v}}$, which of the model parameters will get larger updates? Why might this help with learning?

i, ii <Adam> $\theta \leftarrow \theta - \alpha \odot \frac{m}{\sqrt{v}}$ → ① 손실함수 변화량은 크게 하면서 (positive effect),
→ ② 손실함수 변동량은 작게하는 (negative effect) θ 을 학습해나간다.

$\sqrt{v}$ 는 정규화(normalization) 효과를 가진다.

(b) (4 points) Dropout[3] is a regularization technique. During training, dropout randomly sets units in the hidden layer $\mathbf{h}$ to zero with probability p_{drop} (dropping different units each minibatch), and then multiplies $\mathbf{h}$ by a constant γ. We can write this as

$$\mathbf{h}_{\text{drop}} = \gamma \mathbf{d} \circ \mathbf{h}$$

where $\mathbf{d} \in \{0, 1\}^{D_h}$ (D_h is the size of $\mathbf{h}$) is a mask vector where each entry is 0 with probability p_{drop} and 1 with probability $(1 - p_{\text{drop}})$. γ is chosen such that the expected value of $\mathbf{h}_{\text{drop}}$ is $\mathbf{h}$:

$$\mathbb{E}_{p_{\text{drop}}}[\mathbf{h}_{drop}]_i = h_i$$

for all $i \in \{1, \ldots, D_h\}$.

i. (2 points) What must γ equal in terms of p_{drop}? Briefly justify your answer.

$\gamma = \frac{1}{p_{drop}}$ p_{drop}에 해당하는 뉴런을 꺼버리면 그만큼 상쇄해주어야 하기 때문에 해당 층의 총합을 유지하기 위해 뉴런들을 scaling 한다.
(사실 잘 모르겠다..)

ii. (2 points) Why should we apply dropout during training but not during evaluation?

dropout은 학습과정에서 과적합을 방지하는 규제기법으로, evaluation 과정에는 적용되지 않는다.

2. Neural Transition-Based Dependency Parsing (42 points)

In this section, you'll be implementing a neural-network based dependency parser, with the goal of maximizing performance on the UAS (Unlabeled Attachemnt Score) metric.

Before you begin please install PyTorch 1.0.0 from `https://pytorch.org/get-started/locally/` with the `CUDA` option set to `None`. Additionally run `pip install tqdm` to install the tqdm package – which produces progress bar visualizations throughout your training process.

A dependency parser analyzes the grammatical structure of a sentence, establishing relationships between *head* words, and words which modify those heads. Your implementation will be a *transition-based* parser, which incrementally builds up a parse one step at a time. At every step it maintains a *partial parse*, which is represented as follows:

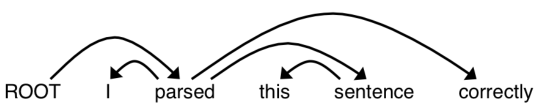
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

Initially, the stack only contains ROOT, the dependencies list is empty, and the buffer contains all words of the sentence in order. At each step, the parser applies a *transition* to the partial parse until its buffer is empty and the stack size is 1. The following transitions can be applied:

- `SHIFT`: removes the first word from the buffer and pushes it onto the stack.
- `LEFT-ARC`: marks the second (second most recently added) item on the stack as a dependent of the first item and removes the second item from the stack.
- `RIGHT-ARC`: marks the first (most recently added) item on the stack as a dependent of the second item and removes the first item from the stack.

On each step, your parser will decide among the three transitions using a neural network classifier.

(a) (6 points) Go through the sequence of transitions needed for parsing the sentence *"I parsed this sentence correctly"*. The dependency tree for the sentence is shown below. At each step, give the configuration of the stack and buffer, as well as what transition was applied this step and what new dependency was added (if any). The first three steps are provided below as an example.

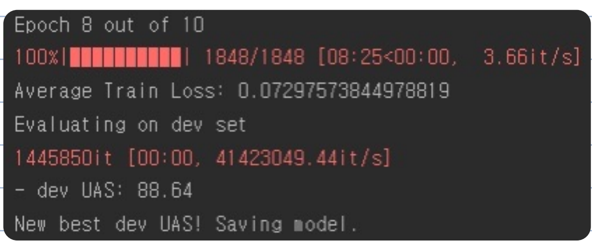


Stack	Buffer	New dependency	Transition
[ROOT]	[I, parsed, this, sentence, correctly]		Initial Configuration
[ROOT, I]	[parsed, this, sentence, correctly]		`SHIFT`
[ROOT, I, parsed]	[this, sentence, correctly]		`SHIFT`
[ROOT, parsed]	[this, sentence, correctly]	parsed→I	`LEFT-ARC`
[ROOT, parsed, this]	[sentence, correctly]		Shift
[ROOT, parsed, this, sentence]	[correctly]		Shift
[ROOT, parsed, sentence]	[correctly]	sentence → this	Left-Arc
[ROOT, parsed]	[correctly]	parsed → sentence	Right-Arc
[ROOT, parsed, correctly]	[ ]		Shift
[ROOT, parsed]	[ ]	parsed → correctly	Right-Arc
[ROOT]	[ ]	ROOT → parsed	Right-Arc

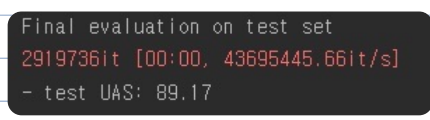
(b) (2 points) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain why.

$2n = n$ (shift) $+ n$ (left/right-arc)

(e)



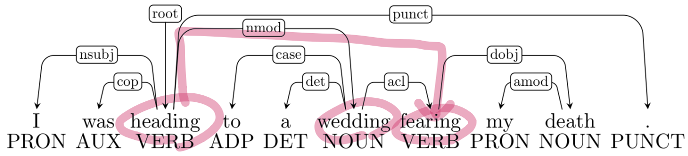
```
Epoch 8 out of 10
100%|██████████| 1848/1848 [08:25<00:00,  3.66it/s]
Average Train Loss: 0.07297573844978819
Evaluating on dev set
1445850it [00:00, 41423049.44it/s]
- dev UAS: 88.64
New best dev UAS! Saving model.
```



```
Final evaluation on test set
2919736it [00:00, 43695445.66it/s]
- test UAS: 89.17
```

(f)

i.

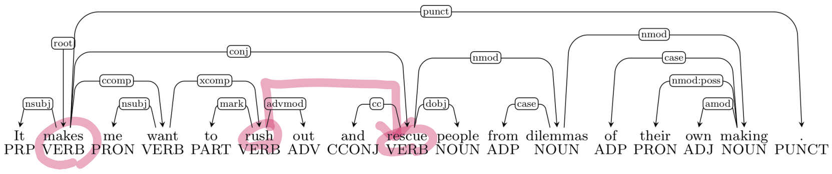


Verb phrase attachment error

Incorrect dependency: wedding → fearing

Correct dependency: heading → fearing

ii.

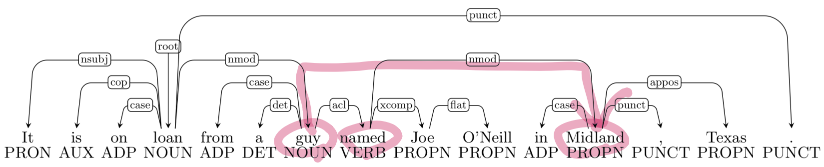


Coordination Attachment Error

Incorrect dependency: makes → rescue

Correct dependency: rush → rescue

iii.

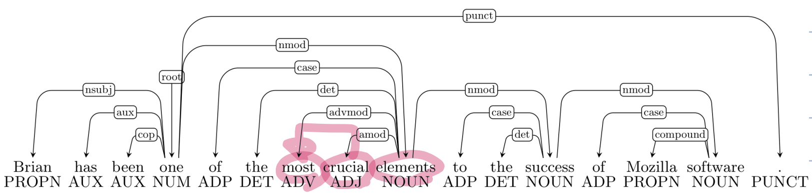


Prepositional Phrase Attachment Error

Incorrect dependency: named → midland

Correct dependency: guy → midland

iv.



Modifier Attachment error

Incorrect dependency: elements → most

Correct dependency: crucial → most