CS224n Assignment #3: Dependency Parsing

1. Machine Learning & Neural Networks (8 points)

(a) (4 points) Adam Optimizer

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

where θ is a vector containing all of the model parameters, J is the loss function, $\nabla_\theta J_{\text{minibatch}}(\theta)$ 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 uses a more sophisticated update rule with two additional steps.

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

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J_{\text{minibatch}}(\theta)$$

$$\theta \leftarrow \theta - \alpha \mathbf{m}$$

where β_1 is a hyperparameter between 0 and 1 (often set to 0.9). Briefly explain in 2-4 sentences (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.

💡 **Answer:**
Adam book-keeps a moving/rolling average (i.e., performs exponential smoothing) of the loss function's gradients, known as momentum, denoted as $\mathbf{m}$. New gradient information is added to the momentum to some extent, controlled by a parameter denoted as β_1. It accomplishes this by multiplying β_1 to the previous momentum, $\mathbf{m}_{i-1}$ and apportions $(1-\beta_1)$ for the latest gradient value. Because β_1 is often set to 0.9, the previous gradient value is valued more than the latest gradient value. In other words, each update step will only vary the newly calculated $\mathbf{m}$ by a little (simply because only 10% of the latest gradient value is added). This results in lower variance and ultimately, a smoother transition from one update to the next, which can potentially make it faster to reach a local optimum (i.e., the training process may be smoother and faster).

Note: With $\beta_1 = 0$, we revert back to Stochastic Gradient Descent (SGD).

One intuition that this may be helpful for learning is that the dampening of oscillations (as a result of lower variance) helps smoothen out the gradient update "trial". That is, the gradient update path no longer "jumps around" hapzardly in a zig-zag pattern when moving towards a local optimum. This can ultimately lead to faster convergence.

Another intuition is that the rolling average is like computing the gradient over a larger minibatch, so each update will be closer to the true gradient over the entire dataset (i.e., lower variance means each gradient estimate is closer to the mean).

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

$$\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1 - \beta_1) \nabla_\theta J_{\text{minibatch}}(\theta)$$

$$\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1 - \beta_2)(\nabla_\theta J_{\text{minibatch}}(\theta) \odot \nabla_\theta J_{\text{minibatch}}(\theta))$$

$$\theta \leftarrow \theta - \alpha \mathbf{m} / \sqrt{\mathbf{v}}$$

where $\odot$ and $/$ denote elementwise multiplication and division (so $z \odot z$ 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?

💡 **Answer:**

Adam normalizes gradient updates (or momentum, $\mathbf{m}$) by $\sqrt{\mathbf{v}}$ where $\mathbf{v}$ is a rolling average of the magnitudes of gradients. This magnifies the magnitude of small components of $\mathbf{m}$ ($\ll 1$) and decreases the magnitude of large components. Therefore, this can give model parameters with small gradients larger updates.

This can help get stagnant parameters to move off of flat areas of the loss landscape (where gradients have plateued or close to 0). Therefore, leading to quicker convergence.

(b) (4 points) Dropout 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:

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

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 $\mathbf{h}$:

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

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

i. (2 points) What must γ equal in terms of p_{drop}? Briefly justify your answer or show your math derivation using the equations given above.

💡 **Answer:**

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

$$= p_{\text{drop}} \times 0 + (1 - p_{\text{drop}}) \times \gamma h_i$$

$$= (1 - p_{\text{drop}})\gamma h_i$$

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

$$(1 - p_{\text{drop}})\gamma h_i = h_i$$

$$(1 - p_{\text{drop}})\gamma = 1$$

$$\boxed{\gamma = \frac{1}{1 - p_{\text{drop}}}}$$

Per "*Dropout: A Simple Way to Prevent Neural Networks from Overfitting" by Srivastava et al., "The weights obtained from pretraining should be scaled up by a factor of* $1/p$ *(where* p *is*
the probability of retaining a unit). This makes sure that for each unit, the expected output from it under random dropout will be the same as the output during pretraining."

During training we drop units at a probability p_{drop}, resulting in probability $p_{\text{keep}} = 1 - p_{\text{drop}}$ of units being retained. By scaling up the units by $\gamma = \frac{1}{1-p_{\text{drop}}} = \frac{1}{p_{\text{keep}}}$, we enable both the training and testing phase to share similar expected outputs.

ii. (2 points) Why should dropout be applied during training? Why should dropout NOT be applied during evaluation? (Hint: it may help to look at the paper linked above in the write-up.)

💡 **Answer:**
Dropout can be interpreted as a regularizer, and a regularizer is aimed at reducing overfitting. When a large neural network is trained on a small training set, it usually performs well on the training data but poorly on held-out test data, indicating that the model has overfitted the training data. When we use dropout during training, we make it harder for the model to overfit to the training data because it only has access to a proportion of its units during a single training iteration.

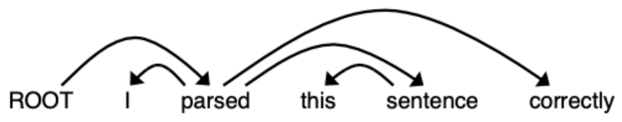
Another way to look at dropout is an ensemble learning method that combines many different weaker classifiers, just like a random forest classifier is composed of various tree classifiers. Each time you randomly drop out a proportion of your units, it’s akin to getting a different randomly-selected model. Using dropout during training forces each of these randomly-selected models to perform well on the task since each classifier is trained individually to some extent (determined by p_{drop}) and learns to tackle a different aspect of the problem. The full model (with no dropout, used for evaluation) can be thought of as an ensemble of these randomly-selected models. Note that during evaluation, dropout should not be applied since we're looking to leverage the learned experience from all of these individual classifiers.

You can also think of dropout in terms of “co-adaptation” as in *Hinton et al.'s* original paper on dropout, *"Improving neural networks by preventing co-adaptation of feature detectors"*, 2012. The paper mentioned that dropout "prevents complex co-adaptations in which a feature detector is only helpful in the context of several other specific feature detectors. Instead, each neuron learns to detect a feature that is generally helpful for producing the correct answer given the combinatorially large variety of internal contexts in which it must operate."

Another way to look at this is that the end goal of the training process is to come up with a set of weights/parameters that generalize well to unseen data. When we dropout units, we're creating different versions of the network by "thinning" out the network. This prevents any single neuron from dominating the output. However, during evaluation, we'd like all neurons to contribute to the output, and each neuron is already trained with the prevention of overfitting in mind so there's no need to do additional dropout during inference time. If we were to apply dropout during evaluation time, we wouldn't be able to fairly assess the generalization power of the network as doing so brings uncertainty to predictions.

2. Neural Transition-Based Dependency Parsing (44 points)

(a) (4 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`

💡 **Answer:**

Stack	Buffer	New dependency	Transition
[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 in 1-2 sentences why.

💡 **Answer:**
A sentence with n words will be parsed in $2n$ steps. Every word in the buffer needs to be pushed on the stack which would take n steps. Eventually, each word has to be popped from the stack to form a dependency which would take another n steps.

(e) (12 points) We are now going to train a neural network to predict, given the state of the stack,
buffer, and dependencies, which transition should be applied next. First, the model extracts a feature vector representing the current state. We will be using the feature set presented in the original neural dependency parsing paper: *A Fast and Accurate Dependency Parser using Neural Networks*. The function extracting these features has

been implemented for you in `utils/parser_utils.py`. This feature vector consists of a list of tokens (e.g., the last word in the stack, first word in the buffer, dependent of the second-to-last word in the stack if there is one, etc.). They can be represented as a list of integers $w = [w_1, w_2, ..., w_m]$ where m is the number of features and each $0 \leq w_i < |V|$ is the index of a token in the vocabulary ($|V|$ is the vocabulary size). Then our network looks up an embedding for each word and concatenates them into a single input vector:

$$\mathbf{x} = [\mathbf{E}_{w_1}, ..., \mathbf{E}_{w_m}] \in \mathbb{R}^{dm}$$

where $\mathbf{E} \in \mathbb{R}^{|V| \times d}$ is an embedding matrix with each row $\mathbf{E}_w$ as the vector for a particular word w.

We then compute our prediction as:

$$\mathbf{h} = \mathrm{ReLU}(\mathbf{xW} + \mathbf{b}_1)$$

$$\mathbf{l} = \mathbf{hU} + \mathbf{b}_2$$

$$\hat{\mathbf{y}} = \mathrm{softmax}(l)$$

where $\mathbf{h}$ is referred to as the hidden layer, $\mathbf{l}$ is referred to as the logits, $\hat{\mathbf{y}}$ is referred to as the predictions, and $\mathrm{ReLU}(z) = \max(z, 0)$. We will train the model to minimize the cross-entropy loss:

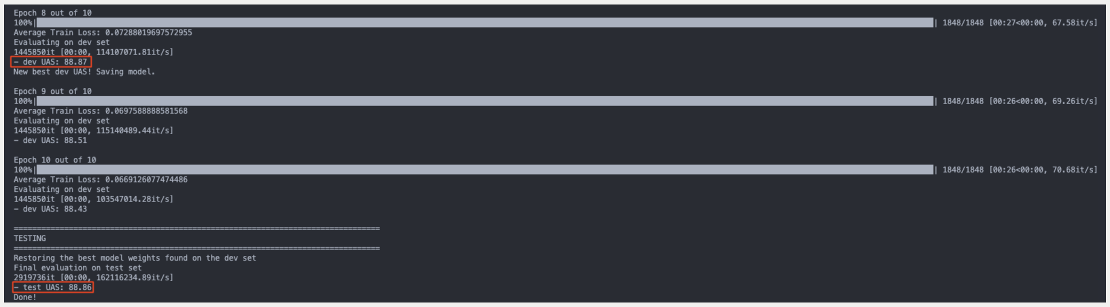
$$J(\theta) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i$$

To compute the loss for the training set, we average this $J(\theta)$ across all training examples.
We will use UAS score as our evaluation metric. UAS refers to Unlabeled Attachment Score, which is computed as the ratio between number of correctly predicted dependencies and the number of total dependencies despite of the relations (our model doesn't predict this).

In parser `model.py` you will find skeleton code to implement this simple neural network using PyTorch. Complete the `__init__` , `embedding_lookup` and `forward` functions to implement the model. Then complete the `train_for_epoch` and `train` functions within the `run.py` file. Finally execute `python run.py` to train your model and compute predictions on test data from Penn Treebank (annotated with Universal Dependencies).

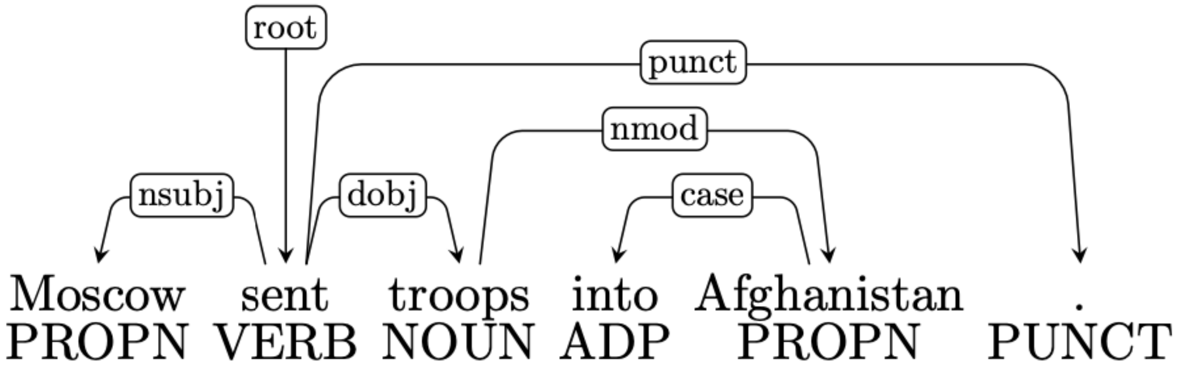
💡 Report of best UAS model:

dev UAS	test UAS
88.87	88.86

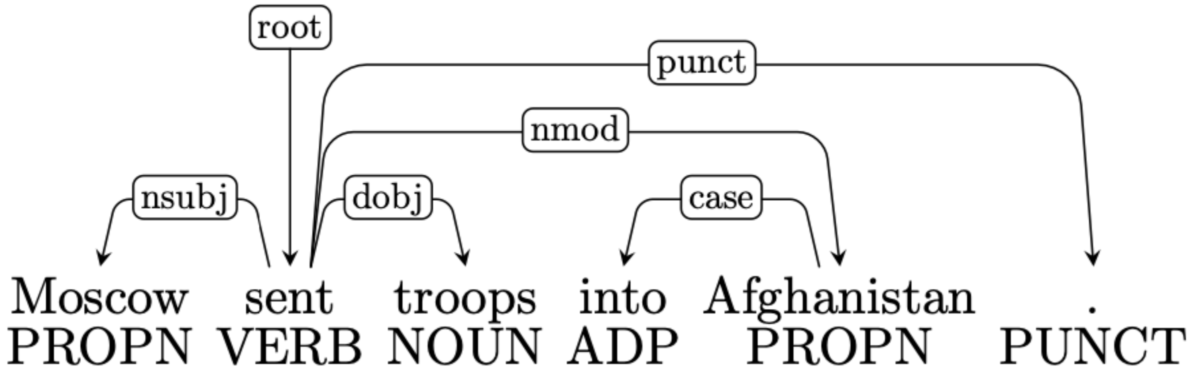


```
Epoch 8 out of 10
100%|████████████████████████████████████████| 1848/1848 [00:27<00:00, 67.58it/s]
Average Train Loss: 0.07288019697572955
Evaluating on dev set
1445850it [00:00, 114107071.81it/s]
- dev UAS: 88.87
New best dev UAS! Saving model.

Epoch 9 out of 10
100%|████████████████████████████████████████| 1848/1848 [00:26<00:00, 69.26it/s]
Average Train Loss: 0.0697588888581568
Evaluating on dev set
1445850it [00:00, 115140489.44it/s]
- dev UAS: 88.51

Epoch 10 out of 10
100%|████████████████████████████████████████| 1848/1848 [00:26<00:00, 70.68it/s]
Average Train Loss: 0.0669126077474486
Evaluating on dev set
1445850it [00:00, 103547014.28it/s]
- dev UAS: 88.43

================================================================================
TESTING
================================================================================
Restoring the best model weights found on the dev set
Final evaluation on test set
2919736it [00:00, 162116234.89it/s]
- test UAS: 88.86
Done!
```

(f) (12 points) We'd like to look at example dependency parses and understand where parsers like ours might be wrong. For example, in this sentence:



the dependency of the phrase into *Afghanistan* is wrong, because the phrase should modify *sent* (as in *sent* into *Afghanistan*) not troops (because *troops* into *Afghanistan* doesn't make sense). Here is the correct parse:



More generally, here are four types of parsing error:

- **Prepositional Phrase Attachment Error:** In the example above, the phrase into *Afghanistan* is a prepositional phrase. A Prepositional Phrase Attachment Error is when a prepositional phrase is attached to the wrong head word (in this example, *troops* is the wrong head word and *sent* is the correct head word). More examples of prepositional phrases include *with a rock*, *before midnight* and *under the carpet*.
- **Verb Phrase Attachment Error:** In the sentence *Leaving the store unattended, I went outside to watch the parade*, the phrase *leaving the store unattended* is a verb phrase. A Verb Phrase Attachment Error is when a verb phrase is attached to the wrong head word (in this example, the correct head word is *went*).
- **Modifier Attachment Error:** In the sentence *I am extremely short*, the adverb *extremely* is a modifier of the adjective *short*. A Modifier Attachment Error is when a modifier is attached to the wrong head word (in this example, the correct head word is *short*).
- **Coordination Attachment Error:** In the sentence *Would you like brown rice or garlic naan?*, the phrases *brown rice* and *garlic naan* are both conjuncts and the word *or* is the coordinating conjunction. The second conjunct (here *garlic naan*) should be attached to the first conjunct (here *brown rice*). A Coordination Attachment Error is when the second conjunct is attached to the wrong head word

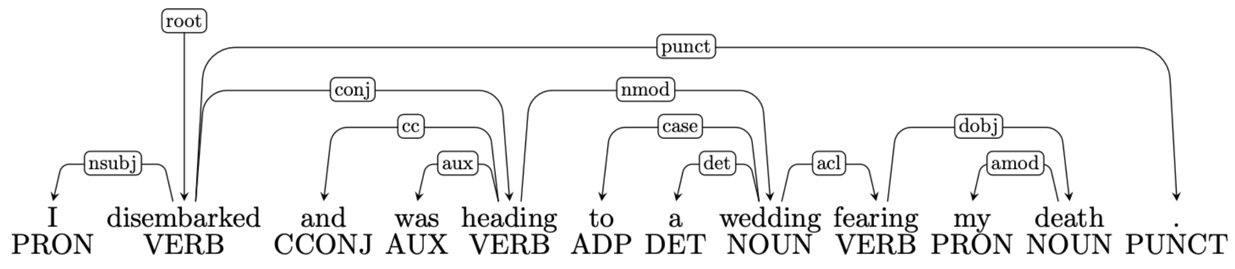
(in this example, the correct head word is *rice*). Other coordinating conjunctions include *and*, *but* and *so*.

In this question are four sentences with dependency parses obtained from a parser. Each sentence has one error type, and there is one example of each of the four types above. For each sentence, state the type of error, the incorrect dependency, and the correct dependency. While each sentence should have a unique error type, there may be multiple possible correct dependencies for some of the sentences. To demonstrate: for the example above, you would write:

- **Error type**: Prepositional Phrase Attachment Error
- **Incorrect dependency**: troops → Afghanistan
- **Correct dependency**: sent → Afghanistan

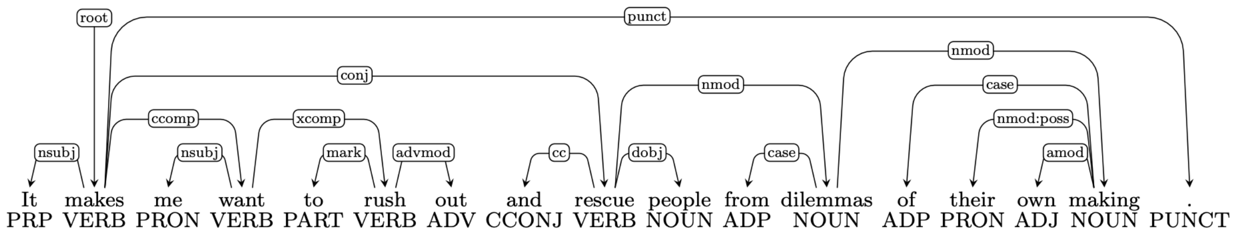
Note: *There are lots of details and conventions for dependency annotation. If you want to learn more about them, you can look at the UD website: http://universaldependencies. org or the short introductory slides at: http://people.cs.georgetown.edu/nschneid/ p/UD-for-English.pdf. Note that you* ***do not*** *need to know all these details in order to do this question. In each of these cases, we are asking about the attachment of phrases and it should be sufficient to see if they are modifying the correct head. In particular, you* ***do not*** *need to look at the labels on the the dependency edges – it suffices to just look at the edges themselves.*

i.



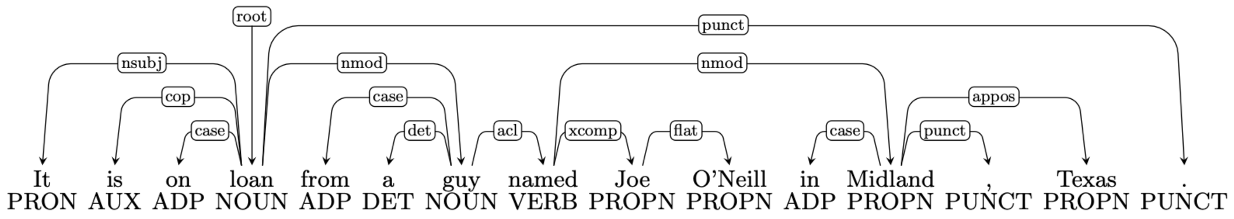
- **Error type:** Verb Phrase Attachment Error
- **Incorrect dependency:** wedding → fearing
- **Correct dependency**: I → fearing

ii.



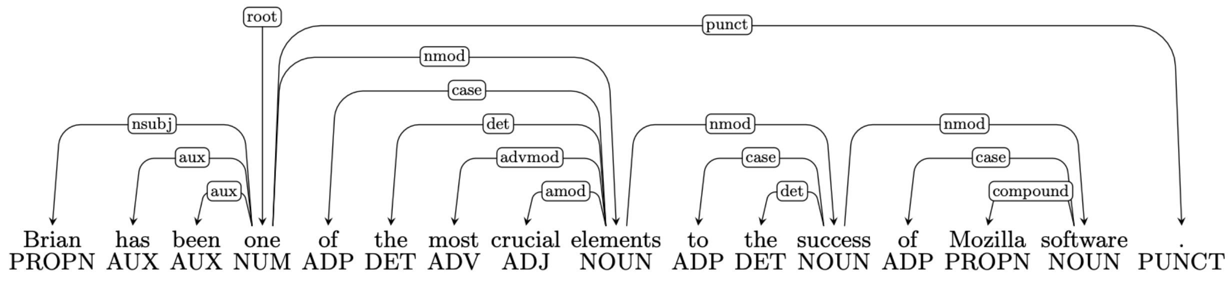
- **Error type:** Coordination Attachment Error
- **Incorrect dependency:** makes → rescue
- **Correct dependency**: rush → makes

iii.



- **Error type**: Prepositional Phrase Attachment Error
- **Incorrect dependency:** named → Midland
- **Correct dependency**: guy → Midland

iv.



- **Error type**: Modifier Attachment Error
- **Incorrect dependency:** elements → most
- **Correct dependency**: crucial → most