CS 224n: Natural Language Processing with Deep Learning **(Due: 01/28/20)**

Submission Assignment #3 (Dependency Parsing (52 Points))

Instructor: Christopher Manning *Name:* Jianpan Gun

In this assignment, you will build a neural dependency parser using PyTorch. In Part 1, you will learn about two general neural network techniques (Adam Optimization and Dropout) that you will use to build the dependency parser in Part 2. In Part 2, you will implement and train the dependency parser, before analyzing a few erroneous dependency parses.

Problem 1: Machine Learning & Neural Networks (4+4=8 points)

(a) (4 points) **Adam Optimizer**
Recall the standard Stochastic Gradient Descent update rule:

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \tag{1}$$

where θ 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 uses a more sophisticated update rule with two additional steps.

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

$$\begin{aligned} &\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \\ &\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \mathbf{m} \end{aligned} \tag{2}$$

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 $\boldsymbol{m}$ stops the updates from varying as much and why this low variance may be helpful to learning, overall.

A: The parameter $\boldsymbol{m}$ store the history gradient and use it to smooth the gradient volatility. Exponential weighted average equal to average the front $t \propto \beta_1$ values. This mechanism makes the speed in the dimension with constant gradient direction faster, and the update speed in the dimension with changed gradient direction slower, so that it can speed up convergence and reduce oscillation. Recursive derivation equation (5), get:

$$\begin{aligned} \mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)g_t = \beta_1[\beta_1 \mathbf{m}_{t-2} + (1-\beta_1)g_{t-1}] + (1-\beta_1)g_t &= \beta_1^2 \mathbf{m}_{t-2} + (1-\beta_1)\beta_1 g_{t-1} + (1-\beta_1)g_t \\ &= \beta_1^3 \mathbf{m}_{t-3} + (1-\beta_1)\beta_1^2 g_{t-2} + (1-\beta_1)\beta_1 g_{t-1} + (1-\beta_1)g_t \\ &= \cdots \\ &= \beta_1^t \mathbf{m}_0 + (1-\beta_1)\sum_{k=0}^{t-1}[\beta_1^k g_{t-k}] \end{aligned} \tag{3}$$

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

$$\begin{aligned} &\mathbf{m} \leftarrow \beta_1 \mathbf{m} + (1-\beta_1)\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \\ &\mathbf{v} \leftarrow \beta_2 \mathbf{v} + (1-\beta_2)(\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta}) \odot \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}(\boldsymbol{\theta})) \\ &\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \odot \mathbf{m}/\sqrt{\mathbf{v}} \end{aligned} \tag{4}$$

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

A: Similarly,

$$\begin{aligned}\mathbf{m}_t &= \beta_1^t \mathbf{m}_0 + (1-\beta_1)\sum_{k=0}^{t-1}\beta_1^k g_{t-k} \approx (1-\beta_1)\sum_{k=0}^{t-1}\beta_1^k g_{t-k}\\ \mathbf{v}_t &= \beta_2^t \mathbf{v}_0 + (1-\beta_2)\sum_{k=0}^{t-1}\beta_2^k g_{t-k}^2 \approx (1-\beta_2)\sum_{k=0}^{t-1}\beta_2^k g_{t-k}^2\end{aligned} \tag{5}$$

Considering,

$$\alpha_t = \alpha \cdot \sqrt{1-\beta_2^t} / \left(1-\beta_1^t\right) \tag{6}$$

So,

$$\begin{aligned}\Delta\theta_t = \theta_t - \theta_{t-1} = -\alpha_t \cdot m_t/(\sqrt{v_t}+\hat{\epsilon}) &= -\alpha\frac{1-\beta_1}{1-\beta_1^t}\sqrt{\frac{1-\beta_2^t}{1-\beta_2}}\frac{\sum_{k=0}^{t-1}\beta_1^k g_{t-k}}{\epsilon + \sqrt{\sum_{k=0}^{t-1}\beta_2^k g_{t-k}^2}}\\ &= -\alpha\mathcal{F}(\beta_1,\beta_2,t)\sum_{k=0}^{t-1}\frac{1}{\epsilon/g_{t_k} + \sqrt{\sum_{i=0}^{t-1}\beta_2^i (g_{t-i}/g_{t_k})^2}}\end{aligned} \tag{7}$$

Due to $g_t < g_{t-i}$. So, the weight of history gradient is higher than before. It also eased the excessively fast gradient descent originally brought by squared gradient and speed up the slow gradient descent.

(b) (4 points) **Dropout is a regularization technique. During training, dropout randomly sets units in the hidden layer h to zero with probability p drop (dropping different units each minibatch), and then multiplies h by a constant γ. We can write this as**

$$\mathbf{h}_{\text{drop}} = \gamma \mathbf{d} \odot \mathbf{h} \tag{8}$$

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

$$\mathbb{E}_{p_{\text{drop}}}\left[\mathbf{h}_{\text{drop}}\right]_i = h_i \tag{9}$$

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

i. (2 points) **What must γ equal in terms of p drop? Briefly justify your answer.**
$\gamma = \frac{1}{1-p_{dropout}}$, due to the expectations of h_{drop} is equal to h_i. So,

$$\mathbb{E}_{p_{\text{drop}}}\left[\mathbf{h}_{\text{drop}}\right]_i = \mathbb{E}_{p_{\text{drop}}}[\gamma\mathbf{d}\odot\mathbf{h}]_i = \gamma\mathbb{E}_{p_{\text{drop}}}[(1-p_{drop})h_i] = \gamma(1-p_{drop})h_i = h_i \tag{10}$$

ii. (2 points) **Why should we apply dropout during training but not during evaluation?**
We need to predict each inputs in the evaluation stage. The `Dropout` will miss some information in the evaluation.

Problem 2: Neural Transition-Based Dependency Parsing

(4+2+6+8+12+12=44 points)

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

Before you begin please install PyTorch 1.4.0 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:

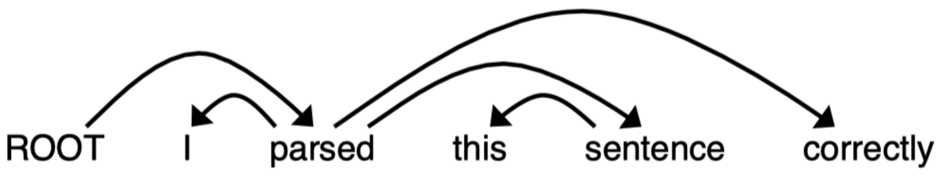
- A *stack* of words that are currently being processed.
- A *buffer* of words yet to be processed.
- A list of *dependencies* predicted by the parser.

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, adding a $first\_word \to second\_word$ dependency to the dependency list.
- 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, adding a $second\_word \to first\_word$ dependency to the dependency list.

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

(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
[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.**

A sentence will be parsed 2n times. First n times are push operation and second n times are pop operation.

(c) (6 points) **Implement the** `__init__` **and** `parse_step` **functions in the** `PartialParse` **class in parser** `transitions.py`**. This implements the transition mechanics your parser will use. You can run basic (non_exhaustive) tests by running** `python parser transitions.py part_c`**.**

Listing 1: PartialParse class.

```
class PartialParse(object):
    ROOT = "ROOT"

    def __init__(self, sentence: List[str]):
        """Initializes this partial parse."""
        self.sentence = sentence
        self.stack = [self.ROOT]
        self.buffer = sentence.copy()
        self.dependencies = []

    def parse_step(self, transition: str):
        """Performs a single parse step by applying the given transition to this ↩
            partial parse

        @param transition (str): A string that equals "S", "LA", or "RA" ↩
            representing the shift,
                                    left-arc, and right-arc transitions. You can assume↩
                                         the provided
                                    transition is a legal transition.
        """
        if transition == "S":
            self.stack.append(self.buffer[0])
            self.buffer = self.buffer[1:]
        else:
            head, tail = self.stack[-2:][:: -1 if transition == "LA" else 1]
            self.dependencies.append((head, tail))
            self.stack.remove(tail)

    def parse(self, transitions):
        for transition in transitions:
            self.parse_step(transition)
        return self.dependencies
```

(d) (8 points) **Our network will predict which transition should be applied next to a partial parse. We could use it to parse a single sentence by applying predicted transitions until the parse is complete. However, neural networks run much more efficiently when making predictions about batches of data at a time (i.e., predicting the next transition for any different partial parses simultaneously). We can parse sentences in minibatches with the following algorithm.**

Algorithm 1 Minibatch Dependency Parsing

```
Input: sentences, a list of sentences to be parsed and model, our model that makes parse decisions.
1: Initialize partial parses as a list of PartialParses, one for each sentence in sentences;
2: Initialize unfinished parses as a shallow copy of partial parses;
3: while unfinished parses is not empty do
4:     Take the first batch size parses in unfinished parses as a minibatch;
5:     Use the model to predict the next transition for each partial parse in the minibatch;
6:     Perform a parse step on each partial parse in the minibatch with its predicted transition;
7:     Remove the completed (empty buffer and stack of size 1) parses from unfinished parses;
8: end while
```

Implement this algorithm in the `minibatch_parse` **function in** `parser_transitions.py`**. You can run basic (non-exhaustive) tests by running** `python parser_transitions.py part_d`**. Note: You will need minibatch parse to be correctly implemented to evaluate the model you will build in part (e). However, you do not need it to train the model, so you should be able to complete most of part (e) even if minibatch parse is not implemented yet.**

Listing 2: minibatch_parse.

```
def minibatch_parse(sentences: List[List[str]], model, batch_size: int):
    """Parses a list of sentences in minibatches using a model."""
    dependencies = []
    partial_parses = [PartialParse(sentence) for sentence in sentences]
    unfinished_parses = partial_parses[:]
    while unfinished_parses:
        minibatch_data = unfinished_parses[:batch_size]
        transitions = model.predict(minibatch_data)
        for pp, transition in zip(minibatch_data, transitions):
            pp.parse_step(transition)
            if len(pp.buffer) == 0 and len(pp.stack) == 1:
                unfinished_parses.remove(pp)
    dependencies = [pp.dependencies for pp in partial_parses]
    return dependencies
```

(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 =** $[w_1, w_2, \ldots, w_m]$ **where m is the number of features and each** $0 \le w_i < |V|$ **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:**

$$\mathbf{x} = [\mathbf{E}_{w_1}, \ldots, \mathbf{E}_{w_m}] \in \mathbb{R}^{dm} \tag{11}$$

where $E \in \mathbb{R}^{|V|d}$ **is an embedding matrix with each row E w as the vector for a particular word w. We then compute our prediction as:**

$$\begin{aligned} \mathbf{h} &= \text{ReLU}(\mathbf{xW} + \mathbf{b}_1) \\ \mathbf{l} &= \mathbf{hU} + \mathbf{b}_2 \\ \hat{\mathbf{y}} &= \text{softmax}(l) \end{aligned} \tag{12}$$

where h is referred to as the hidden layer, l is referred to as the logits, ŷ is referred to as the predictions, and ReLU(z) = max(z, 0)). We will train the model to minimize cross-entropy loss:

$$J(\theta) = CE(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{i=1}^{3} y_i \log \hat{y}_i \tag{13}$$

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).

Note:

- For this assignment, you are asked to implement Linear layer and Embedding layer. Please DO NOT use torch.nn.Linear or torch.nn.Embedding module in your code, otherwise you will receive deductions for this problem.
- Please follow the naming requirements in our TODO if there are any, e.g. if there are explicit requirements about variable names you have to follow them in order to receive full credits. You are free to declare other variable names if not explicitly required.

Hints:

- Once you have implemented embedding lookup (e) or forward (f) you can call python parser model.py with flag -e or -f or both to run sanity checks with each function. These sanity checks are fairly basic and passing them doesn't mean your code is bug free.

- When debugging, you can add a debug flag: python run.py -d. This will cause the code to run over a small subset of the data, so that training the model won't take as long. Make sure to remove the -d flag to run the full model once you are done debugging.
- When running with debug mode, you should be able to get a loss smaller than 0.2 and a UAS larger than 65 on the dev set (although in rare cases your results may be lower, there is some randomness when training).
- It should take about 1 hour to train the model on the entire the training dataset, i.e., when debug mode is disabled.
- When debug mode is disabled, you should be able to get a loss smaller than 0.08 on the train set and an Unlabeled Attachment Score larger than 87 on the dev set. For comparison, the model in the original neural dependency parsing paper gets 92.5 UAS. If you want, you can tweak the hyperparameters for your model (hidden layer size, hyperparameters for Adam, number of epochs, etc.) to improve the performance (but you are not required to do so).

Deliverables:

- Working implementation of the neural dependency parser in parser model.py. (We'll look at and run this code for grading).
- Report the best UAS your model achieves on the dev set and the UAS it achieves on the test set.

Listing 3: ParserModel class.

```python

class ParserModel(nn.Module):

    def __init__(self, embeddings, n_features=36,
                 hidden_size=200, n_classes=3, dropout_prob=0.5):
        """ Initialize the parser model. """
        super(ParserModel, self).__init__()
        self.n_features = n_features
        self.n_classes = n_classes
        self.dropout_prob = dropout_prob
        self.embed_size = embeddings.shape[1]
        self.hidden_size = hidden_size
        self.embeddings = nn.Parameter(torch.tensor(embeddings))

        self.embed_to_hidden_weight = nn.Parameter(nn.init.xavier_normal_(
            torch.zeros(self.embed_size * n_features, hidden_size)))
        self.embed_to_hidden_bias = nn.Parameter(nn.init.xavier_normal_(
            torch.zeros(1, hidden_size)))

        self.dropout = nn.Dropout(dropout_prob)

        self.hidden_to_logits_weight = nn.Parameter(nn.init.xavier_normal_(
            torch.zeros(hidden_size, n_classes)))
        self.hidden_to_logits_bias = nn.Parameter(nn.init.xavier_normal_(
            torch.zeros(1, n_classes)))
        self.RELU = nn.functional.relu

    def embedding_lookup(self, w):
        """ Utilize ‘w‘ to select embeddings from embedding matrix  """
        # 1) For each index ‘i‘ in ‘w‘, select ‘i‘th vector from self.embeddings
        # 2) Reshape the tensor using ‘view‘ function if necessary
        ###
        x = self.embeddings[w].view((w.shape[0], -1))
        return x

    def forward(self, w):
        """ Run the model forward."""
        x = self.embedding_lookup(w)

        h = self.RELU(torch.matmul(x,
                                   self.embed_to_hidden_weight) + self.↩
                                      embed_to_hidden_bias)
        logits = torch.matmul(h,
                              self.hidden_to_logits_weight) + self.↩
                                 hidden_to_logits_bias
        return logits
```

- test UAS: **89.21**

(f) (12 points) **In this question are four sentences with dependency parses obtained from a parser. Each sentence has one error, 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. To demonstrate: for the example above, you would write:**

A. I disembarked and was heading to a wedding fearing my death.

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

B. It makes me want to rush out and rescue people from dilemmas.
of their own making

- **Error type:** Coordination Attachment Error
- **Incorrect dependency:** rush → out
- **Correct dependency:** rush → and

C. It is on loan from a guy named Joe O'Neill in Midland , Texas .

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

D. Brian has been one of the most crucial elements to the success of Mozilla software .

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