CS 224n Assignment #2: Dependency Parsing

1. Machine Learning & Neural Networks (8 points)

(a) (4 points) Adam Optimizer

Recall the standard Stochastic Gradient Descent update rule:

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

where θ is a vector containing all of the model parameters, J is the loss function, $\bigtriangledown_\theta J_{\text{minibatch}}(\theta)$ 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.

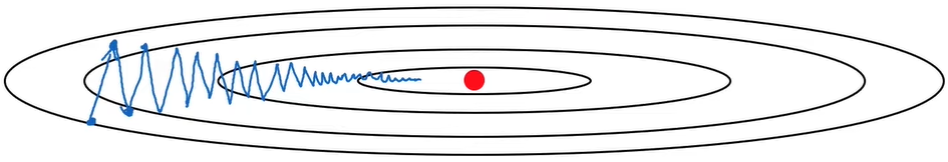
â€‹ i. (2 points) First, Adam uses a trick called `momentum` by keeping track of m, a rolling average of the gradients:

$$m \leftarrow \beta_1 m - (1 - \beta_1) \bigtriangledown_\theta J_{\text{minibatch}}(\theta)$$
$$\theta \leftarrow \theta - \alpha m$$

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

Answer:

In equation $m_{now} \leftarrow \beta_1 m_{previous} - (1 - \beta_1) \bigtriangledown_\theta J_{\text{minibatch}}(\theta)$, $m_{previous}$ can be seen as the velocity of the gradient descent from the previous step, $\bigtriangledown_\theta J_{\text{minibatch}}(\theta)$ can be seen as the acceleration to the gradient at the current step, β_1 can be seen as the friction applied to the previous gradient or the weight on the previous gradient, and $(1 - \beta_1)$ is the weight on the current gradient.



The graph above illustrates the process of a 2-d normal gradient descent. We typically wants to accelerate the horizontal process and punish the vertical process. By weighted summing the previous gradient, the vertical fluctuations tend to cancel out each other and the horizontal process tend of accelerate.

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

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

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:

This is an implementation of RMSprop algorithm. Suppose $\mathbf{v}$ is 2 dimensional, vertically and horizontally (same as the graph in question i).

$$\mathbf{v}_{vertical} \leftarrow \beta_2 \mathbf{v}_{vertical} + (1-\beta_2)\left(\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}\left(\boldsymbol{\theta}\right) \odot \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}\left(\boldsymbol{\theta}\right)\right)$$
$$\mathbf{v}_{horizontal} \leftarrow \beta_2 \mathbf{v}_{horizontal} + (1-\beta_2)\left(\nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}\left(\boldsymbol{\theta}\right) \odot \nabla_{\boldsymbol{\theta}} J_{\text{minibatch}}\left(\boldsymbol{\theta}\right)\right)$$
$$\boldsymbol{\theta}_{vertical} \leftarrow \boldsymbol{\theta}_{vertical} - \alpha \odot \mathbf{m} / \sqrt{\mathbf{v}_{vertical}}$$
$$\boldsymbol{\theta}_{horizontal} \leftarrow \boldsymbol{\theta}_{horizontal} - \alpha \odot \mathbf{m} / \sqrt{\mathbf{v}_{horizontal}}$$

We want $\sqrt{\mathbf{v}_{horizontal}}$ to be relatively small so that $\alpha \odot \mathbf{m} / \sqrt{\mathbf{v}_{horizontal}}$ is large and thus the horizontal process is fast. We want $\sqrt{\mathbf{v}_{vertical}}$ to be relatively large so that $\alpha \odot \mathbf{m} / \sqrt{\mathbf{v}_{vertical}}$ is small and thus the vertical process is slow.

(b) (4 points) Dropout

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 . We can write this as

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

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 $(1-p_{drop})$. γ is chosen such that the expected value of $\mathbf{h}_{drop}$ is $\mathbf{h}$:

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

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

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

Answer:

$$\begin{aligned}
\mathbb{E}_{p_{\text{drop}}}\left[\mathbf{h}_{\text{drop}}\right]_i &= h_i \\
\mathbb{E}_{p_{\text{drop}}}\left[\mathbf{h}_{\text{drop}}\right]_i &= \mathbb{E}_{p_{\text{drop}}}\left[\gamma\mathbf{d} \circ \mathbf{h}\right]_i \\
&= \gamma(1-p_{drop})h_i \\
\gamma(1-p_{drop})h_i &= h_i \\
\gamma &= \frac{1}{1-p_{drop}}
\end{aligned}$$

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

Dropout is a technique to prevent model overfitting to the training data during training. In evaluation, the model is already trained that we don't need the randomness in our prediction. Randomness in evaluation will only make the model performance worse.

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.

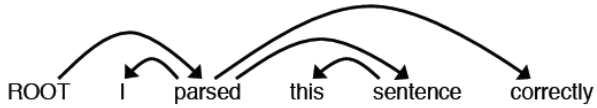
Initially, the stack only contains ROOT, the dependencies list is empty, and the buffer contains all words of the sentence in order. At each step, the parser applies a *transition* to the partial parse until its buffer is empty and the stack size is 1. The following transitions can be applied:

- 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

Answer:

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]	I ← parsed	LEFT-ARC
[ROOT, parsed, this]	[sentence, correctly]		SHIFT
[ROOT, parsed, this, sentence]	[correctly]		SHIFT
[ROOT, parsed, sentence]	[correctly]	this ← sentence	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.

Answer: 2n+1

Initially all n words are in the buffer. We need to perform n steps in total to move n words from the buffer to the Stack. In Stack, for each word, we need to perform one LEFT-ARC or RIGHT-ARC operation to take the one word out from the Stack, which in total has n steps. Therefore, there will be 2n steps plus 1 step for the initial configuration.