Stanford CS224n Assignment 3: Dependency Parsing

Aman Chadha / amanc@stanford.edu

January 31, 2021

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) Adam optimization uses a trick called *momentum* by keeping track of m, a rolling average of the gradients:

$$\begin{aligned} m_i &\leftarrow \beta_1 m_{i-1} + (1-\beta_1)\nabla_\theta J_{minibatch}(\theta) \\ \theta &\leftarrow \theta - \alpha m_i \end{aligned}$$

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 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 and fuses them together to some degree, denoted by β_1, with new gradient information. It accomplishes this by multiplying β_1 to the previous rolling average m_{i-1} and apportions $(1-\beta_1)$ for the most recent gradient value so that changes in the recent past are valued much more (since β_1 is usually set to 0.9) than the newly retrieved gradient at position θ (since $1-\beta_1$ is 0.1). In other words, each update will be mostly the same as the previous one (only a $1-\beta_1$ proportion of m receives an update at each step), so the updates won't vary as much. This results in lower variance and ultimately, smoother transitions from one update to the next, which can potentially maker it faster to reach a local optimum. (As a side note, with $\beta_1 = 0$, we retrieve back simple SGD.)
- One intuition is that the dampening of oscillations as a result of lower variance is helpful for learning because it straightens out the parameter update "trail" that descends into the minimum. Graphically, parameter updates show less of a zig-zag pattern with momentum than without, especially near the minimum. Put simply, momentum helps prevent the model parameters from "bouncing around as much" when mowing towards a local optimum. This can ultimately lead to faster convergence. Another intuition is that doing the rolling average is a bit like computing the gradient over a larger minibatch, so each update will be closer to the true gradient over the whole dataset (i.e., lower variance means each gradient estimate is closer to the mean).

[1] Kingma and Ba, 2015, `ttps://arxiv.org/pdf/1412.6980.pdf`

[2] The actual Adam update uses a few additional tricks that are less important, but we won't worry about them here. If you want to learn more about it, you can take a look at: `http://cs231n.github.io/neural-networks-3/#sgd`

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

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

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 $\sqrt{v}$ which of the model parameters will get larger updates? Why might this help learning?

Answer:

- Adam normalizes parameter updates by maintaining v, a rolling average of the magnitudes of gradients. Adam does so by dividing a smaller amount from parameters with relatively smaller gradients during a parameter update. In other words, entries with small v will get larger updates owing to them being divided by their relatively smaller gradient magnitudes/norms.
- This can be helpful because it can get stagnant parameters move off flat areas of the loss landscape (where gradients are small) and in turn, expedite convergence.

(b) (4 points) Dropout[3] 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:

$$h_{\text{drop}} = \gamma d \odot h$$

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 $(1 - p_{\text{drop}}) \cdot \gamma$ is chosen such that the expected value of h_{drop} is h:

$$E_{p_{\text{drop}}}\left[h_{\text{drop}}\right]_i = h_i$$

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

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

Answer:

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

- Alternatively, we can look at this mathematically as follows,

$$\mathbb{E}_{p_{\text{drop}}}\left[h_{\text{drop}}\right]_i = \mathbb{E}_{p_{\text{drop}}}\left[\gamma d_i h_i\right] = p_{\text{drop}} \times 0 + (1 - p_{\text{drop}})\,\gamma h_i = (1 - p_{\text{drop}})\,\gamma h_i$$

For this to equal h_i, γ must equal $\frac{1}{1-p_{\text{drop}}}$.

[3] Srivastava et al., 2014, https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf

ii. (2 points) Why should dropout be applied during training? Why should dropout NOT be applied during evaluation?

Answer:

There are several possible ways to explain why we apply dropout during training (i.e., the overall purpose of dropout as a technique):

- 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 overfit 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 dropout a proportion of your units, you get 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, as 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 mentions 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."

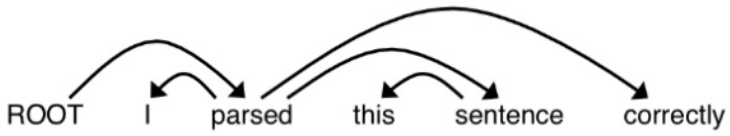
- Another way to look at this is that the end goal of the training process is to come up with a set of weights/parameters that generalize well to unseen data. When we dropout units, we're creating different versions of the network by "thinning" out the network. This prevents any single neuron from dominating the output. However, during evaluation, we'd like all neurons contribute to the output, and each neuron is already trained with preventing overfitting in mind so there's no need to do additional dropping. Thus, 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) Transition-Based Parse: A parser which incrementally builds up a parse one step at a time. At every step it maintains a /partial parse/ which is represented as:

- 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 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.



Answer:

Stack	Buffer	New Dependency	Transition	Step
(ROOT)	[I, parsed, this, sentence, correctly]		Initial Config	0
(ROOT, I)	[parsed, this, sentence, correctly]		SHIFT	1
(ROOT, I, parsed)	[this, sentence, correctly]		SHIFT	2
(ROOT, parsed)	[this, sentence, correctly]	parsed → I	LEFT-ARC	3
(ROOT, parsed, this)	[sentence, correctly]		SHIFT	4
(ROOT, parsed, this, sentence)	[correctly]		SHIFT	5
(ROOT, parsed, sentence)	[correctly]	sentence → this	LEFT-ARC	6
(ROOT, parsed)	[correctly]	parsed → sentence	RIGHT-ARC	7
(ROOT, parsed, correctly)	[]		SHIFT	8
(ROOT, parsed)	[]	parsed → correctly	RIGHT-ARC	9
(ROOT)	[]	ROOT → parsed	RIGHT-ARC	10

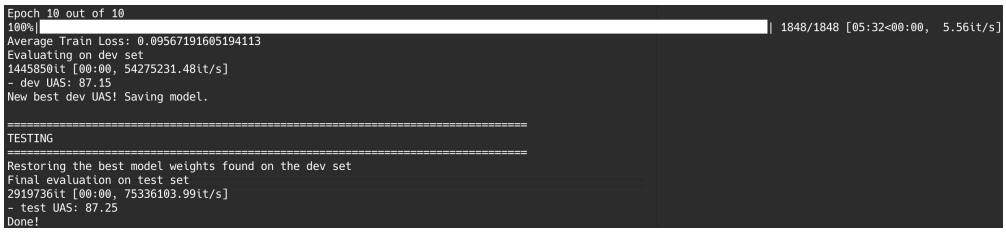
(b) (2 points) A sentence containing n words will be parsed in how many steps (in terms of n)? Briefly explain in 1-2 sentences why.

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) Report of best UAS model:

Answer:

dev UAS	test UAS
87.15	87.25



(f) (12 points) For each sentence state the type of error, the incorrect dependency, and the correct dependency:

Answer:

(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 → rescue

(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