Assignment 3 - Written: Machine Learning & Neural Networks

1. Machine Learning & Neural Networks

(a) Adam Optimizer.

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

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

where β_1 is a hyperparameter between 0 and 1. Briefly explain how using $\mathbf{m}$ stops the update from varying as much and why this low variance may be helpful to learning, overall.

Solution: By maintaining a sort of exponential smoothing, or rolling average, of the loss function's gradients, we effectively control a "memory bank" that tracks previous update steps and fuses them together to some degree, β_1, with new gradient information. This can smooth transitions, i.e. lower variance, from one state to the next when we place the right amount of β_1 on the previous state $\mathbf{m}_{i-1}$ so that recent changes have a bit more importance than the newly retrieved gradient at position θ. Note that when we don't care about the past, $\beta_1 = 0$, we get back simple SGD. This dampening of oscillations is helpful because it straightens out the trail that descends into the minimum. Informally, the efficiency of optimization increases when update steps zig-zag (vary) less.

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

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

Since Adam divides the update by $\sqrt{\mathbf{v}}$ which of the model parameters will get larger updates? Why might this help learning?

Solution: Because $\mathbf{v}$ is a rolling average of the magnitudes of gradients, parameters of the model with corresponding small gradients, i.e. small $\mathbf{v}$ entries, will get larger updates. We're cutting out a smaller amount from such parameters during an update because we divide them by their small gradient magnitude/norm. This might be helpful because it can get recent stagnant parameters moving more efficiently along their axes and in turn expedite convergence.

(b) 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 in each minibatch), and then multiplies $\mathbf{h}$ by a constant γ. We can write this as:

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

where $\mathbf{d} \in \{0,1\}^{D_k}$ is a mask vector.

(i) What must γ equal in terms of p_{drop}?

Solution: During training we drop units at a rate of p_{drop}, resulting in roughly $p_{keep} = 1 - p_{drop}$ fraction of units left over. At test time we'd like to have the effect of keeping a similar fraction, 'p_{keep}, of units on. By scaling down the layer units by $\gamma = \frac{1}{1-p_{drop}} = \frac{1}{p_{keep}}$, we effectively level out their magnitudes so that both phases of learning share very similar expected outputs.

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

Solution: The goal of dropout is to reduce overfitting. We're interested in updating unit weights so as to form a network that performs well across different datasets. Now, during evaluation we're concerned with how well the model handles unseen data. When we dropout units, we're "thinning" out the network which in many cases will add noise to predictions and dampen accuracy. Thus, if we were to apply dropout during evaluation time, we would not be able to fairly assess the generalization power of the network.

2. Neural Transition-Based Dependency Parsing

(a) 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.

Solution:

Stack	Buffer	New Dependency	Transition
(ROOT)	[I, parsed, this, sentence, correctly]		Initial Config
(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) How many steps will it take to parse n words (in terms of n)?
Solution: In the worst case, parsing will take linear time, i.e. $O(n)$. At any step of parsing, we have two possible state transitions, either shifting a word from the buffer to the stack or clearing a dependent from the stack. Every word must spend a single step being shifted from the buffer, thus n words cost n shift

steps. From the stack a word must be "arc"-ed over as a dependent exactly once, thus n words cost n "arc"-ing steps. Therefore, we have $2 * n$ steps giving a cost of $O(n)$.

(e) Report of best UAS model:

dev UAS	test UAS
89.60	89.74

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

(i)

- **Error Type**:
- **Incorrect Dependency**:
- **Correct Dependency**:

(ii)

- **Error Type**:
- **Incorrect Dependency**:
- **Correct Dependency**:

(iii)

- **Error Type**:
- **Incorrect Dependency**:
- **Correct Dependency**:

(iv)

- **Error Type**:
- **Incorrect Dependency**:
- **Correct Dependency**: