Assignment 2

1 Tensorflow Sotfmax

1. **Implement the softmax function using TensorFlow**

Our approach is almost identical to the numpy implementation in Assignment 1. Here, `softmax_1d` handles the case for a 1-D vector, and `tf.map_fn` applies that function to each row of a tensor.

```
def softmax_1d(x):
  x = tf.exp(x - tf.reduce_max(x))
  s = tf.reduce_sum(x)
  return x/s
out = tf.map_fn(lambda _: loss(_), x)
```

2. **Implement the cross-entropy loss using TensorFlow**

Make sure to convert `y` into `dtype float`. `tf.multiply` is element-wise multiplication of tensors, so `multiply` followed by `reduce_sum` is the dot-product.

```
y = tf.to_float(y)
out = -1*tf.reduce_sum(tf.multiply(y, tf.log(yhat)))
```

3. **Explain the purpose of placeholder variables and feed dictionaries in TensorFlow computations**

Placeholder variables are analogous to X in the equation $y = X\theta + b$. Like X, placeholder variables represent the input data to the algorithm. The feed dictionary substitutes actual values for the placeholder variables.

2 Neural Transition-Based Dependency Parser

1. **Step-through transitions to prase "I parsed this sentence correctly"**

Stack	Buffer	New dependency	Transition
`[ROOT]`	`[I,parsed,this,sentence,correctly]`	`NA`	`NA`
`[ROOT,I]`	`[parsed,this,sentence,correctly]`	`NA`	`SHIFT`
`[ROOT,I,parsed]`	`[this,sentence,correctly]`	`NA`	`SHIFT`
`[ROOT,parsed]`	`[this,sentence,correctly]`	`parsed->I`	`LEFT-ARROW`
`[ROOT,parsed, this]`	`[sentence,correctly]`	`NA`	`SHIFT`
`[ROOT,parsed,this,sentence]`	`[correctly]`	`NA`	`SHIFT`
`[ROOT,parsed,sentence]`	`[correctly]`	`sentence->this`	`LEFT-ARROW`
`[ROOT,parsed]`	`[correctly]`	`parsed->sentence`	`RIGHT-ARROW`
`[ROOT,parsed,correctly]`	`[]`	`NA`	`SHIFT`
`[ROOT,parsed]`	`[]`	`parsed->correctly`	`LEFT-ARROW`

2. **How many steps to parse sentence with n words?**

Each word has to be added to and removed from the stack once, so parsing a sentence with n words will take $2n$ steps.

6. **Derive the value of the constant multiplier in dropout**

To regularlize via dropout, we randomly set units in a hidden layer h to 0, and then multiply each unit by a constant γ. The value for a unit h_i after this operation is $h_{drop} = \gamma p_{drop} h_i$. What value should we choose for γ to maintain the expected value of h_i? The expectation of h_{drop} is

$$\gamma \left(0 \cdot p_{drop} + (1 - p_{drop})\right) h_i$$

.

Setting equal to h_i and solving for γ, we get

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

.

7. **The Adam optimizer**

7.1. By setting the update parameters to be the (weighted) average of the gradients at the current step, plus the rolling average of the gradients at all previous steps, Adam reduces the influence of the latest gradient values on the update. This slows down the learning rate as training proceeds, thus reducing variation in later trainig epochs.

7.2. Dividing by the rolling average of the magnitude of the gradient has the effect of reducing the impact of gradient components with high magnitudes. This prevents dimensions with high gradient magnitudes from domininating the learning process.

3 Recurrent Neural Newtworks: Language Modeling

1. **Argue equivalence of cross-entropy loss and perplexity**

In $CE\left(\mathbf{y^{(t)}}, \mathbf{\hat{y}^{(t)}}\right)$, only the dimension k corresponding to the correct class is non-zero, so

$$\begin{aligned} -\sum \mathbf{y_j^{(t)}} \log \mathbf{\hat{y}_j^{(t)}} &= -\log \mathbf{\hat{y}_k^{(t)}} \\ &= \log \frac{1}{\mathbf{\hat{y}_k^{(t)}}} \end{aligned}$$

This shows that CE is the *log* of perplexity. Since the *log* is a monotonic function, maximizing CE also maximizes perplexity.