Prof. Dr. Andreas Hotho, Albin Zehe, Konstantin Kobs
Lehrstuhl für Data Science

22./23.06.2021

6. Assignment in "Machine Learning for Natural Language Processing"

Summer Term 2021

1 General Questions

1. a) What are the advantages of LSTM and GRU over Vanilla RNN?

   b) What are the differences between LSTM and GRU?

a) Vanilla RNNs often suffer from the exploding and vanishing gradient problem. LSTM solve this problem by using gating mechanisms, preventing the exponentiation of the weight matrix when backpropagating over multiple timesteps.

b) GRU is a simpler variation of RNN than LSTM. Most notably, it uses fewer gates. In practice, LSTM and GRU tend to work similarly well.

? Something to think about

2. In Neural Machine Translation, the goal is to translate a sentence in a source language to an equivalent sentence in a different target language.

   Assume we want to use a simple RNN/LSTM to translate the sentences "Ich habe keine Zeit" and "Er schwieg eine Zeit lang" from German to English, by running them through the network and outputting a target word after every timestep.

   Identify some problems of this approach! Can you think of ways to address these problems?

Sometimes, it is important to consider parts of a sentence that only appear after the current word in order to find a correct translation. In the example, "Zeit" would be translated to "time" in the first sentence and to "while" in the second sentence. Another problem is that the sentence structure may be very

different in the target language, meaning that the word order must change in the translation.

Both of these problems can be fixed by first reading the entire sentence into an encoder and then decoding it into the target language. Sometimes it is a good idea to use a bi-directional RNN (a combination of one RNN reading over the sentence from left to right and one RNN reading from right to left), to have access to the entire sentence at every timestep.

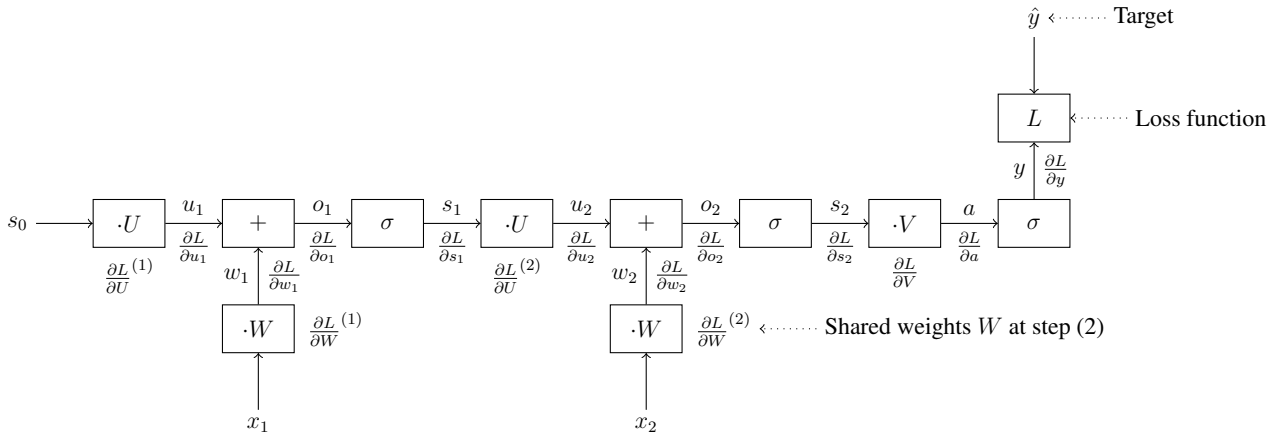
More on this soon in the lecture!

2 Recurrent Neural Networks

This task is about getting more practice in applying the Backpropagation algorithm. Below, you find a flow graph defining an unrolled recurrent neural network.

Remember that BPTT is only Backpropagation on an unrolled recurrent neural network!

The network takes sequences as input of length $l = 2$. Every element in the sequence shall be a row vector $x_i \in \mathbb{R}^{1\times 3}$. For the sake of simplicity, the target value of the network is a scalar $\hat{y}, y \in \mathbb{R}$ and the internal state has a size of 2, that is $s_i \in \mathbb{R}^{1\times 2}$. Furthermore, we use $L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$ as a loss function. We use sigmoid as the activation function.



1. You can infer the shape of all learnable parameters from the above information. What are the shapes of U, W and V?

   a) Since U projects the previous state $s_{i-1} \in \mathbb{R}^2$ into the current state space and $s_i \in \mathbb{R}^{1\times 2}$, it follows that $U \in \mathbb{R}^{2\times 2}$

b) We know that $u_i = s_{i-1}U \in \mathbb{R}^{1\times 2}$. In order to be able to compute $o_i = u_i + w_i$, we need w_i to be of size $\mathbb{R}^{1\times 2}$ and since $w_i = x_i W$ and $x_i \in \mathbb{R}^{1\times 3}$, it follows that $W \in \mathbb{R}^{3\times 2}$.

c) Finally, we know that $s_i \in \mathbb{R}^{1\times 2}$ and the output $y \in \mathbb{R}$, which means V is in $\mathbb{R}^{2\times 1}$.

2. Derive the formula for every partial derivative annotated in the graph above by applying backpropagation. Remember that the matrices W and U are shared between inputs x_i. The partial derivatives for these matrices have been annotated with the step i, e.g. $\frac{\partial L}{\partial W}^{(i)}$, to help differentiate between steps i.

Derivative of the loss function:

$$\frac{\partial L}{\partial y} = \frac{1}{2} \cdot 2 \cdot (\hat{y} - y) \cdot (-1) = y - \hat{y}$$

Derivatives of the fully connected layer:

$$\begin{aligned}
\frac{\partial L}{\partial a} &= \frac{\partial L}{\partial y}\frac{\partial y}{\partial a} = \frac{\partial L}{\partial y}\sigma(a)(1-\sigma(a)) \\
\frac{\partial L}{\partial s_2} &= \frac{\partial L}{\partial a}\frac{\partial a}{\partial s_2} = \frac{\partial L}{\partial a}V^T \\
\frac{\partial L}{\partial V} &= \frac{\partial L}{\partial a}\frac{\partial a}{\partial V} = s_2^T\frac{\partial L}{\partial a}
\end{aligned}$$

Derivatives of the second RNN step:

$$\frac{\partial L}{\partial o_2} = \frac{\partial L}{\partial s_2}\frac{\partial s_2}{\partial o_2} = \frac{\partial L}{\partial s_2}\sigma(o_2)(1-\sigma(o_2))$$
$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial o_2}\frac{\partial o_2}{\partial w_2} = \frac{\partial L}{\partial o_2}\frac{\partial(u_2+w_2)}{\partial w_2} = \frac{\partial L}{\partial o_2}\cdot 1$$
$$\frac{\partial L}{\partial W}^{(2)} = \frac{\partial L}{\partial w_2}\frac{\partial w_2}{\partial W} = x_2^T\frac{\partial L}{\partial w_2}$$
$$\frac{\partial L}{\partial u_2} = \frac{\partial L}{\partial o_2}\frac{\partial o_2}{\partial u_2} = \frac{\partial L}{\partial o_2}\frac{\partial(u_2+w_2)}{\partial u_2} = \frac{\partial L}{\partial o_2}\cdot 1$$
$$\frac{\partial L}{\partial s_1} = \frac{\partial L}{\partial u_2}\frac{\partial u_2}{\partial s_1} = \frac{\partial L}{\partial u_2}U^T$$
$$\frac{\partial L}{\partial U}^{(2)} = \frac{\partial L}{\partial u_2}\frac{\partial u_2}{\partial U} = s_1^T\frac{\partial L}{\partial u_2}$$

Derivatives of the first RNN step:

$$\frac{\partial L}{\partial o_1} = \frac{\partial L}{\partial s_1}\frac{\partial s_1}{\partial o_1} = \frac{\partial L}{\partial s_1}\sigma(o_1)(1-\sigma(o_1))$$
$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial o_1}\frac{\partial o_1}{\partial w_1} = \frac{\partial L}{\partial o_1}\frac{\partial(u_1+w_1)}{\partial w_1} = \frac{\partial L}{\partial o_1}\cdot 1$$
$$\frac{\partial L}{\partial W}^{(1)} = \frac{\partial L}{\partial w_1}\frac{\partial w_1}{\partial W} = x_1^T\frac{\partial L}{\partial w_1}$$
$$\frac{\partial L}{\partial u_1} = \frac{\partial L}{\partial o_1}\frac{\partial o_1}{\partial u_1} = \frac{\partial L}{\partial o_1}\frac{\partial(u_1+w_1)}{\partial u_1} = \frac{\partial L}{\partial o_1}\cdot 1$$
$$\frac{\partial L}{\partial U}^{(1)} = \frac{\partial L}{\partial u_1}\frac{\partial u_1}{\partial U} = s_0^T\frac{\partial L}{\partial u_1}$$

3. Calculate all partial derivatives based on the following inputs:

$$x_1 = \begin{pmatrix} 1.0 & 0.5 & 2.0 \end{pmatrix}, x_2 = \begin{pmatrix} 0.1 & 0.8 & 1.0 \end{pmatrix}, \hat{y} = 1$$

$$U = \begin{pmatrix} 0.1 & 0.8 \\ 0.2 & 0.5 \end{pmatrix}, W = \begin{pmatrix} 0.1 & 0.8 \\ 0.2 & 0.5 \\ 0.4 & 1.0 \end{pmatrix}, V = \begin{pmatrix} 0.5 \\ 0.3 \end{pmatrix}$$

$$s_0 = \begin{pmatrix} 0.0 & 0.0 \end{pmatrix}$$

You can use Python/Numpy to do the actual calculations.

```
import numpy as np


def sigmoid(x):
    return np.exp(x) / np.sum(np.exp(x) + 1)


def L(x, xt):
    return 0.5 * (xt - x)**2


x = np.array([[1.0, 0.5, 2.0], [0.1, 0.8, 1.0]],
             dtype=np.float32)
yt = np.array([1])

U = np.array([[0.1, 0.8], [0.2, 0.5]], dtype=np.float32)
W = np.array([[0.1, 0.8], [0.2, 0.5], [0.4, 1.0]],
             dtype=np.float32)
V = np.array([0.5, 0.3], dtype=np.float32)

# initial state = 0.0
s0 = np.array([0.0, 0.0], dtype=np.float32)

# extract inputs from sequence
x1 = x[0]
x2 = x[1]

# forward pass
```

```
# t=1
u1 = s0 @ U
w1 = x1 @ W
o1 = u1 + w1
s1 = sigmoid(o1)
# t=2
u2 = s1 @ U
w2 = x2 @ W
o2 = u2 + w2
s2 = sigmoid(o2)

# fcn
a = s2 @ V
y = sigmoid(a)

print("y={}".format(y))
print("Loss={}".format(L(y, yt)))

# backward pass
dy = y - yt
da = dy * sigmoid(a) * (1 - sigmoid(a))
ds2 = da * V.T
dV = s2.T * da

# t=2
do2 = ds2 * sigmoid(o2) * (1 - sigmoid(o2))
dw2 = do2
dW2 = np.outer(x2.T, dw2)
du2 = do2
ds1 = du2 @ U.T
dU2 = np.outer(s1.T, du2)

# t=1
do1 = ds1 * sigmoid(o1) * (1 - sigmoid(o1))
dw1 = do1
dW1 = np.outer(x1.T, dw1)
du1 = do1
ds0 = du1 @ U.T
dU1 = np.outer(s1.T, du1)
```

```
print(dV)
print(dU1)
print(dU2)
print(dW1)
print(dW2)
```

3 Python

3.1 Implementing Neural Networks Part 5 — Embeddings

In this assignment, you will implement a neural network "library" yourself, using `python` and `numpy`.

This week, you will use your implementation to train your own word embeddings on a small corpus.

For this, download the dataset of English sentences from `http://mattmahoney.net/dc/text8.zip` and extract the .txt file.

1. Extract training samples from the given corpus: Slide a context window over the corpus, creating tuples of the form $\big(\underbrace{w_{i+2}}_{\text{center word}}, \underbrace{(w_i, w_{i+1}, w_{i+3}, w_{i+4})}_{\text{context words}} \big)$
2. Implement a simplified version of the skip-gram model: Your model should have the following layers:
   - Input: a 1-hot vector representing the center word of each sample
   - Hidden Layer: a layer of size 32
   - Output: a layer of size $|V|$, where V is the vocabulary of the corpus
3. Train the network on your samples: Use the center word as the input and train the network to predict the "4-hot" vector of the context words, that is, a vector that is zero everywhere except at the positions corresponding to the output words. (Note that this is not how the original Word2Vec is calculated. We change the procedure as you have not implemented the Cross Entropy loss function.)
4. After training, where will the embeddings be found?

Feel free to try different optimisers, activation functions, layer sizes, maybe apply Dropout, and look into the resulting word embeddings!