CS187 Lab 2-2 – Recurrent neural networks

September 5, 2023

[ ]:
```
# Please do not change this cell because some hidden tests might depend on it.
import os

# Otter grader does not handle ! commands well, so we define and use our
# own function to execute shell commands.
def shell(commands, warn=True):
    """Executes the string `commands` as a sequence of shell commands.

       Prints the result to stdout and returns the exit status.
       Provides a printed warning on non-zero exit status unless `warn`
       flag is unset.
    """
    file = os.popen(commands)
    print (file.read().rstrip('\n'))
    exit_status = file.close()
    if warn and exit_status != None:
        print(f"Completed with errors. Exit status: {exit_status}\n")
    return exit_status

shell("""
ls requirements.txt >/dev/null 2>&1
if [ ! $? = 0 ]; then
 rm -rf .tmp
 git clone https://github.com/cs187-2021/lab2-2.git .tmp
 mv .tmp/tests ./
 mv .tmp/requirements.txt ./
 rm -rf .tmp
fi
pip install -q -r requirements.txt
""")
```

[ ]:
```
# Initialize Otter
import otter
grader = otter.Notebook()
```

You've read about recurrent neural networks (RNN), but there's nothing like carrying out the calculations yourself to really understand what's going on in these systems.

In this lab, you'll carry out RNN calculations by hand or by programming – both forward (calculating outputs from inputs) and backward (computing gradients) – and use RNNs to simulate bigram language models.

New bits of Python used for the first time in the *solution set* for this lab, and which you may therefore find useful:

- `torch.clamp`: restricts all elements of a tensor to a specific range
- `torch.diag`: creates a tensor with the given inputs as the diagonal
- `torch.eye`: creates an identity matrix
- `torch.mv` (typically invoked via the `@` operator): matrix-vector multiplication
- `torch.prod`: takes the product of elements in a vector
- `torch.T`: returns the transpose of a tensor
- `torch.zeros`: creates a matrix of zeros

Preparation – Loading packages

[ ]:
```
import sys
import torch
import wget
import math

# Script for visualizing computation graphs
data_dir = "data/"
sys.path.append(data_dir)
os.makedirs(data_dir, exist_ok=True)

wget.download('https://raw.githubusercontent.com/nlp-course/data/master/scripts/
 ↪makedot.py', out=data_dir)
from makedot import make_dot
```

1 The forward step

A *recurrent neural network* (RNN) works by calculating a sequence of hidden states $\mathbf{h} = \langle h_0, \ldots, h_N \rangle$ and outputs $\mathbf{o} = \langle o_1, \ldots, o_N \rangle$ from a sequence of inputs $\mathbf{x} = \langle x_1, \ldots, x_N \rangle$. (We're notating the elements, like x_t, h_t, and o_t as if they are scalars, but you should keep in mind that they might well be vectors themselves.)

Each hidden state h_t and output o_t of an RNN is calculated from an input x_t and the previous hidden state h_{t-1} using a set of weights $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ according to the following equations. (We ignore all bias terms for simplicity, and due to the fact that in large neural networks they make no big difference.)

$$h_t = \sigma(\mathbf{U}x_t + \mathbf{V}h_{t-1}) \quad \text{hidden state} \tag{1}$$
$$o_t = \mathbf{W}h_t \quad \text{output score} \tag{2}$$
$$\hat{y}_t = \sigma(o_t) \quad \text{predicted output distribution} \tag{3}$$

The figure at right may be helpful in giving a picture of the RNN computation.

Question: To check your understanding, notice that we've defined **h** so that it has one more element than **x** and **o**. Why is this?

Type your answer here, replacing this text.

1.1 Defining the RNN and its input

To better understand this process, we set up an example of some RNN parameters and values for x_1 and x_2 in the next cell, so that you can carry out the RNN operations yourself to calculate h_1, h_2, o_1, and o_2. (In this example, the length of the input N is 2, and the dimensionality of the x, h, and o vectors are also 2. Thus, $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ are all 2×2 matrices.)

[ ]:
```
# RNN parameters
U = torch.Tensor([ [-0.3,  0.6],
                   [ 0.2,  0.1] ])
V = torch.Tensor([ [ 0.4,  0.4],
                   [ 0.9, -0.7] ])
W = torch.Tensor([ [ 0.2,  0.1],
                   [-0.2,  0.5] ])

# inputs
x1 = torch.Tensor([0.0, 1.0])
x2 = torch.Tensor([0.3, 0.4])

# initial value for the hidden state, a zero vector
h0 = torch.Tensor([0, 0])

# Set which nodes to visualize for later
visualized_nodes = [U, V, W, x1, x2, h0]
for p in visualized_nodes:
    p.requires_grad = True
```

1.2 Carrying out the forward step

Given these definitions, calculate values for h_1, h_2, o_1, and o_2 using `torch` operations.

You may assume for this problem that the nonlinearity σ used in calculating h_t is a Rectified Linear Unit (ReLU), defined by

$$\text{ReLU}(x) = \max(0, x)$$

ReLU can be implemented as `torch.clamp(x, min=0)`

[ ]:
```
def relu(x):
    return torch.clamp(x, min=0)

# TODO: calculate h1, h2, o1, and o2 using torch operations
...
h1 = ...
o1 = ...
h2 = ...
o2 = ...
```

[ ]:
```
grader.check("forward")
```

We print out the results for manual verification

[ ]:
```
print (f"o1:\t{o1} \n"
       f"o2:\t{o2}")
```

1.3 Visualizing the computation graph

Now, we can visualize the computation graph, that is, the graph of how o_1 and o_2 are computed from other variables and parameters.

Note that to make the below code work, you need to install `graphviz`:

- MacOS: `brew install graphviz`
- Deepnote: already installed
- Google Colab: already installed
- Ubuntu: `sudo apt-get install graphviz`

On Deepnote, the below code might not show anything, but you can check the generated PDF in `computation_graph.pdf`.

[ ]:
```
# Define the mapping from variable name to variable,
# so that the nodes in our computation graph can be labeled
params = {k: eval(k) for k in ['U', 'V', 'W',
                               'x1', 'x2',
                               'h0',
                               'h1', 'o1',
                               'h2', 'o2']}

# Visualize the computation graph constructed by PyTorch
dot = make_dot((o1, o2), params=params)

# Save Graph to `computation_graph.pdf`
```

```
dot.render(data_dir + 'computation_graph')

# Visualize (not working in Deepnote)
dot
```

Is the generated computation graph what you expected?

2 Expressing bigram language models via RNNs

In this section, as an exercise in understanding how RNNs work, you'll design an RNN to behave like a bigram model. By providing a reduction from bigram models to RNNs, you thereby show that RNNs are (not surprisingly) more expressive than bigram language models.

2.1 Bigram language models

Recall that a bigram language model uses the previous word to predict the next word in a sequence $w_1 \cdots w_N$. A bigram model over a vocabulary $\mathbf{v} = \{v_1, \ldots, v_V\}$ is specified by a set of probabilities $\Pr(w_{t+1} = v_j \,|\, w_t = v_i)$, the probability that a word of type v_j follows a word of type v_i. (As usual, we'll abbreviate this probability $\Pr(v_j \,|\, v_i)$, since the probability is assumed to be the same for all t.)

We can pack these probabilities into a single table T with V rows and V columns such that

$$T_{ij} = \Pr(v_j \,|\, v_i)$$

Importantly, the sum $\sum_{j=1}^{V} T_{ij}$ is 1 for all i. The vector T_i (the i-th row of table T) thus constitutes the probability distribution for the word following v_i.

For this activity, we use a vocabulary with only two tokens (we can think of them as a and b) encoded as one-hot vectors (v_1 = `[1,0]` and v_2 = `[0,1]`) and we provide a transition table T. We also define two sample sequences representing *aaaaaabbaa* and *babbbbabbb*.

[ ]:
```
# the vocabulary V
Vocab = torch.Tensor([ [1, 0],
                       [0, 1] ])

# the bigram probabilities T_ij
T = torch.Tensor([ [0.6, 0.4],
                   [0.3, 0.7] ])

# two sample sequences
seq1 = torch.Tensor(
    [ [1,0], [1,0], [1,0], [1,0], [1,0], [1,0], [0,1], [0,1], [1,0], [1,0] ])
seq2 = torch.Tensor(
    [ [0,1], [1,0], [0,1], [0,1], [0,1], [0,1], [1,0], [0,1], [0,1], [0,1] ])
```

Before proceeding, take a guess as to which of the two sequences would be more likely according to the provided bigram model. (We won't hold you to the guess.)

Now write a function `sequence_probability` to find the probability of a sequence according to the bigram model. Below, we'll use that function to check your guess.

> As in the previous lab, we'll ignore the contribution of the first $n-1$ tokens (the first token in the case of this example), since it doesn't have sufficient context. In a full n-gram model, we'd pad the start of the string with $n-1$ "start of sequence" tokens that are not part of the main vocabulary and have probability 1. It's standard also to add an "end of sequence" token at the end as well. But we'll pass on these niceties for now, if for no other reason than that it would double the size of the vocabulary you'd have to deal with.

[ ]:
```
# TODO -- Write a function to find the probability of a sequence given a bigram table
def sequence_probability(seq, T):
    """Returns the probability of a sequence `seq` under a bigram table `T`.
    Arguments:
      seq: a sequence of size N x vocab_size, where seq[i] is the one-hot
           representation for the i-th word.
      T: a bigram table, where T_{ij} = P(v_j | v_i).
    Returns: the probability of the sequence, a tensor of a single element."""
    ...
```

[ ]:
```
grader.check("bigram")
```

We can use the `sequence_probability` function to find the probabilities of the two sequences.

[ ]:
```
print(f"Probability of A: {sequence_probability(seq1, T):.5f}\n"
      f"Probability of B: {sequence_probability(seq2, T):.5f}")
```

Was your guess correct?

2.2 Tiny bigram RNN

In theory, given enough capacity, an RNN can approximate any function arbitrarily well. (In practice, we don't have infinite capacity, and even if we did have sufficient capacity, that doesn't mean that a particular optimization method, like stochastic gradient descent as you've been using in project segment 1, can find the global optimum.)

In this section, you'll show that an RNN is at least as expressive as a bigram language model. You'll design the activation function σ and the parameters $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$ of an RNN such that it behaves exactly like the particular bigram model above, by taking as input a sequence of one-hot representations of words, and outputting at each step a vector with the probabilities of the next word.

For instance, the probabilities for the word following v_i should be T_i, the row from the transition matrix. Given a sequence beginning with v_1, represented by the one-hot encoding [1, 0], your RNN's first output should be

$$o_1 = \mathbf{W}h_1 = \mathbf{W}\sigma\left(\mathbf{U}\begin{bmatrix}1\\0\end{bmatrix} + \mathbf{V}h_0\right) = T_1$$

etc. Assume that h_0, the initial h value, is $\begin{bmatrix}0\\0\end{bmatrix}$.

Hint 1: Use $\sigma(x) = x$ for simplicity.

Hint 2: You're going to want to work this out on paper before filling in your solution in the next cell.

[ ]:
```
#TODO -- Define the parameters of the model in terms of the bigram probability␣
↪matrix `T` and other constants
h0 = torch.zeros(2)
sigma = lambda x: x
U = ...
V = ...
W = ...
```

[ ]:
```
grader.check("tiny_bigram_RNN")
```

2.3 Consensus section: General bigram RNN

This section is more open-ended in nature and need only be turned in for the consensus submission of the lab.

Above, you worked out the solution for implementing a bigram model for a two-word vocabulary as an RNN. But the construction can be generalized for bigram models over vocabularies of arbitrary size, say, N word types.

Question: In general, given an $N \times N$ bigram probability matrix T_{ij} what would be the activation function σ and the parameters $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{W}$, of an RNN that outputs T_i given v_i as an input?

Type your answer here, replacing this text.

3 The backward step

The remainder of the lab needs you to take some derivatives. In these exercises you will both build your intuition and gain a concrete understanding of how back-propagation works in an RNN.

Back-propagation is the method used to compute the gradients of an RNN. First, the RNN runs forward on some inputs, and then we calculate the loss as a function of the output (a measure of how far off the RNN's output o_t was from the desired output y_t). The loss function we will be using is the squared error:

$$L = \sum_{t=1}^{N} (y_t - o_t)^2$$

For the particular case of a language model, the "desired output" is just the next word in the sequence, so we have

$$L = \sum_{t=1}^{N} (x_{t+1} - o_t)^2$$

> There's a little issue with the last step, since x_{N+1} doesn't exist. Let's pretend that x_{N+1} exists below for brevity of notations. (See the comment above about end-of-sequence tokens.)
>
> Note that we use squared error loss here mainly for simplicity. In real language modeling tasks people use cross entropy loss, which you've seen in project segment 1.

We then find the derivatives of the loss with respect to each parameter and use the derivatives to make small adjustments to the parameters to reduce the loss. This process is repeated until the loss is minimized.

To minimize the loss function, we need to calculate the derivative of the loss L with respect to all parameters. In this lab, we only consider how to calculate the derivative of L with respect to $\mathbf{U}$, but other parameters work similarly.

For simplicity, let's assume for now that $h_0, \ldots, h_N$, $x_1, \ldots, x_N$ and $o_1, \ldots, o_N$ are all scalars. Therefore, the parameters $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$ are all scalars as well. Such an assumption avoids taking gradients of vectors and matrices, although the below results can be easily generalized.

In the next few subsections, you'll derive the gradient formulas for RNNs operating on sequences first of length 1, then 2, then 3, and finally on arbitrary length sequences.

3.1 RNN backprop on very very short sequences

Consider an RNN run on an input sequence of length 1. This RNN's output sequence will therefore consist only of a single output o_1.

As you can see from the loss function given above, $L = (x_2 - o_1)^2$ is a function of o_1 and x_2, therefore we can find

$$\frac{\partial L(o_1, x_2)}{\partial o_1} = 2(o_1 - x_2)$$

Furthermore, o_1 is a function of h_1 and $\mathbf{W}$, so we can find $\frac{\partial o_1(h_1, \mathbf{W})}{\partial h_1}$. Finally, h_1 is a function of $\mathbf{U}$, $\mathbf{V}$, h_0, and x_1, so we can find $\frac{\partial h_1(\mathbf{U}, \mathbf{V}, h_0, x_1)}{\partial \mathbf{U}}$.

First, let's visualize the computation graph for more intuition. Note that different from the previous computation graph, we only show the variables that we are interested in.

[ ]:
```
# RNN parameters
U_ = torch.Tensor([0.1]) # we use U_ instead of U (etc.) to avoid autograding␣
 ↪issues
V_ = torch.Tensor([0.2])
W_ = torch.Tensor([0.3])

# inputs
x1_ = torch.Tensor([0.5])
x2_ = torch.Tensor([0.6])
x3_ = torch.Tensor([0.7])

# initial value for the hidden state, a zero vector
h0_ = torch.Tensor([0])

# Set which nodes to visualize later
visualized_nodes = [U_, ]
for p in visualized_nodes:
    p.requires_grad = True

# Calculate h1, o1
h1_ = relu(U_ * x1_ + V_ * h0_)
o1_ = W_ * h1_

L_ = (o1_ - x2_) ** 2

params = {k: eval(k+'_') for k in ['L', 'U', 'h1', 'o1']}
dot = make_dot(L_, params=params)
# Save Graph to `computation_graph_1.pdf`
dot.render(data_dir + 'computation_graph_1')
dot
```

Question: Find a formula for $\frac{\partial L(\mathbf{U},\mathbf{V},\mathbf{W},x_1,x_2,h_0)}{\partial \mathbf{U}}$ in terms of $\frac{\partial L(o_1,x_2)}{\partial o_1}$, $\frac{\partial o_1(\mathbf{W},h_1)}{\partial h_1}$,and $\frac{\partial h_1(\mathbf{U},\mathbf{V},h_0,x_1)}{\partial \mathbf{U}}$. You might find the above computation graph useful: how does L depend on $\mathbf{U}$?

> Note that in your answer you can omit the arguments of the functions for brevity (e.g., you can use $\frac{\partial L}{\partial o_1}$ instead of $\frac{\partial L(o_1,x_2)}{\partial o_1}$), but keep in mind that the arguments are important: e.g., o_1 eventually depends on $\mathbf{U}$, so $\frac{\partial o_1(\mathbf{U},\mathbf{V},\mathbf{W},x_1,x_2,h_0)}{\partial \mathbf{U}}$ is likely not zero, but if instead we assume its arguments are h_1 and $\mathbf{W}$, i.e., $o_1 = o_1(h_1, \mathbf{W})$, then $\frac{\partial o_1}{\partial U}$ is undefined or 0, as when we take partial derivatives, we hold other arguments (h_1 and $\mathbf{W}$ in this case) as constant.
>
> Hint: Use the chain rule.

Type your answer here, replacing this text.

3.2 RNN backprop on very short sequences

Consider an RNN run on an input sequence of length 2. This RNN's output sequence o will consist of o_1 and o_2 and the loss will be

$$L = (x_2 - o_1)^2 + (x_3 - o_2)^2$$

Let's visualize the computation graph.

[ ]:
```
# RNN parameters
U_ = torch.Tensor([0.1])
V_ = torch.Tensor([0.2])
W_ = torch.Tensor([0.3])

# inputs
x1_ = torch.Tensor([0.5])
x2_ = torch.Tensor([0.6])
x3_ = torch.Tensor([0.7])

# initial value for the hidden state, a zero vector
h0_ = torch.Tensor([0])

# Set which nodes to visualize later
visualized_nodes = [U_, ]
for p in visualized_nodes:
    p.requires_grad = True

# Calculate h1, o1
h1_ = relu(U_ * x1_ + V_ * h0_)
h2_ = relu(U_ * x2_ + V_ * h1_)
o2_ = W_ * h2_
o1_ = W_ * h1_

L_ = (o1_ - x2_)**2 + (o2_ - x3_)**2

params = {k: eval(k+'_') for k in ['L', 'U', 'h1', 'o1', 'h2', 'o2']}

dot = make_dot(L_, params=params)
# Save Graph to `computation_graph_2.pdf`
dot.render(data_dir + 'computation_graph_2')
dot
```

Question: This time, find the formula for $\frac{\partial L(\mathbf{U},\mathbf{V},\mathbf{W},x_1,x_2,x_3,h_0)}{\partial \mathbf{U}}$, the derivative of the loss L with respect to the parameter $\mathbf{U}$ in terms of $\frac{\partial L(o_1,o_2,x_2,x_3)}{\partial o_t}$, $\frac{\partial o_t(\mathbf{W},h_t)}{\partial h_t}$, $\frac{\partial h_2(\mathbf{U},\mathbf{V},h_1,x_2)}{\partial h_1}$, and $\frac{\partial h_t(\mathbf{U},\mathbf{V},h_{t-1},x_t)}{\partial \mathbf{U}}$ for $t \in \{1, 2\}$. You might find the above computation graph useful.

> How many possible paths are there in the computation graph from $\mathbf{U}$ to L? Each path corresponds to a term in the final answer.

Hint: when we take $\frac{\partial h_2(\mathbf{U},\mathbf{V},h_1,x_2)}{\partial \mathbf{U}}$, we are holding other arguments (other than $\mathbf{U}$) as constants, such as h_1. Therefore, $\frac{\partial h_2}{\partial \mathbf{U}}$ does not reflect the path through h_1 where $\mathbf{U}$ can also exert influence on h_2: $\mathbf{U} \to h_1 \to h_2$.

Type your answer here, replacing this text.

3.3 RNN backprop on short sequences

For the penultimate backprop challenge, consider an RNN run on an input sequence of length 3. This RNN's output sequence o will consist of o_1, o_2, and o_3.

Let's look at the computation graph. How many possible paths are there from $\mathbf{U}$ to L? How many h elements are there in each path?

[ ]:
```
# RNN parameters
U_ = torch.Tensor([0.1])
V_ = torch.Tensor([0.2])
W_ = torch.Tensor([0.3])

# inputs
x1_ = torch.Tensor([0.5])
x2_ = torch.Tensor([0.6])
x3_ = torch.Tensor([0.7])
x4_ = torch.Tensor([0.8])

# initial value for the hidden state, a zero vector
h0_ = torch.Tensor([0])

# Set which nodes to visualize later
visualized_nodes = [U_, ]
for p in visualized_nodes:
    p.requires_grad = True

# Calculate h1, o1
h1_ = relu(U_ * x1_ + V_ * h0_)
h2_ = relu(U_ * x2_ + V_ * h1_)
h3_ = relu(U_ * x3_ + V_ * h2_)
o3_ = W_ * h3_
o2_ = W_ * h2_
o1_ = W_ * h1_

L_ = (o1_ - x2_)**2 + (o2_ - x3_)**2 + (o3_ - x4_)**2

params = {k: eval(k+'_') for k in ['L', 'U', 'h1', 'o1', 'h2', 'o2', 'h3',␣
 ↪'o3']}

dot = make_dot(L_, params=params)
# Save Graph to `computation_graph_3.pdf`
```

```
dot.render(data_dir + 'computation_graph_3')
dot
```

Question: This time, find the formula for $\frac{\partial L(\mathbf{U},\mathbf{V},\mathbf{W},x_1,x_2,x_3,x_4,h_0)}{\partial \mathbf{U}}$ in terms of $\frac{\partial L(o_1,o_2,o_3,x_2,x_3,x_4)}{\partial o_t}$, $\frac{\partial o_t(\mathbf{W},h_t)}{\partial h_t}$, $\frac{\partial h_t(\mathbf{U},\mathbf{V},h_{t-1},x_t)}{\partial h_{t-1}}$, and $\frac{\partial h_t(\mathbf{U},\mathbf{V},h_{t-1},x_t)}{\partial \mathbf{U}}$ for $t \in \{1,2,3\}$.

You should start to see a pattern emerging from this solution.

Type your answer here, replacing this text.

3.4 Looking back

In the last few problems your answers were in terms of partial derivatives that we gave you. In this exercise you will calculate part of a partial derivative.

Question: Referring back to the first problem in this lab, recall that

$$\mathbf{U} = \begin{bmatrix} -0.3 & 0.6 \\ 0.2 & 0.1 \end{bmatrix}, \mathbf{V} = \begin{bmatrix} 0.4 & 0.4 \\ 0.9 & -0.7 \end{bmatrix}, x_1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, h_0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Find the partial derivative $\frac{\partial h_1}{\partial U_{1,1}}$, where $U_{1,1}$ is the element in the first row and first column of the matrix $\mathbf{U}$.

Hint: We didn't tell you what the function σ is. It turns out that you will not need it! If you set it up right, you will see that you will not have to do much math at all.

Type your answer here, replacing this text.

Now, let's verify your solution above using PyTorch, which can calculate gradients automatically. We show an example below of how to calculate gradients. Adapt it to find the gradients of $\frac{\partial h_{1,1}}{\partial \mathbf{U}}$ (where $h_{1,1}$ denotes the first element of vector h_1, and here we are taking the derivative with respect to the entire matrix $\mathbf{U}$). Use ReLU as σ.

[ ]:
```
# TODO: modify the code below to find the gradients of
# $\frac{\partial h_{1,1}}{\partial \vect{U}}$

# RNN parameters
U_ = torch.Tensor([ [-0.3,  0.6],
                    [ 0.2,  0.1] ])
V_ = torch.Tensor([ [ 0.4,  0.4],
                    [ 0.9, -0.7] ])
U_.requires_grad = True

# inputs
x1_ = torch.Tensor([0.0, 1.0])

# initial value for the hidden state, a zero vector
h0_ = torch.Tensor([0, 0])
h1_ = ...
```

```
h11 = ...

# The magic gradient computation
h11.backward()
U_grad = U_.grad
print (f'Gradient of W: {U_grad}')
```

[ ]:
```
grader.check("automatic_diff")
```

Does the computed gradient agree with your calculation of $\frac{\partial h_1}{\partial U_{1,1}}$? (Note that here you only calculated $\frac{\partial h_{1,1}}{\partial \mathbf{U}}$ through PyTorch, so let's just focus on $\frac{\partial h_{1,1}}{\partial U_{1,1}}$.)

By now, hopefully you have got some idea of how PyTorch computes gradients with respect to $\mathbf{U}$ by calling a `backward` function on `h11`: the reason is that PyTorch maintains an underlying computation graph, and it can find all ancestors of `h11` (including $\mathbf{U}$). The gradient computation process is essentially applying chain rules on this computation graph. (Think of how you would implement it yourself.)

3.5 Optional section: RNN backprop on arbitrary sequences

This section is more challenging in nature and is therefore completely optional and will not affect your grade.

The final challenge! Consider an RNN run on an input sequence of arbitrary length N.

Question: Find a general formula for $\frac{\partial L(\mathbf{U},\mathbf{V},\mathbf{W},x_1,\cdots,x_{N+1},h_0)}{\partial \mathbf{U}}$ in terms of $\frac{\partial L(o_1,\cdots,o_N,x_2,\cdots,x_{N+1})}{\partial o_t}$, $\frac{\partial o_t(\mathbf{W},h_t)}{\partial h_t}$, $\frac{\partial h_t(\mathbf{U},\mathbf{V},h_{t-1},x_t)}{\partial h_{t-1}}$, and $\frac{\partial h_t(\mathbf{U},\mathbf{V},h_{t-1},x_t)}{\partial \mathbf{U}}$ for $t \in \{1,\ldots,N\}$.

Hint: How many terms are there when N is 1, 2, or 3? How many possible paths are there from $\mathbf{U}$ to L in the computation graph?

Hint: Your solution might look a little like

$$\sum + \sum\left(\sum\left(\prod\right)\right)$$

Type your answer here, replacing this text.

4 Lab debrief

Question: We're interested in any thoughts you have about this lab so that we can improve this lab for later years, and to inform later labs for this year. Please list any issues that arose or comments you have to improve the lab. Useful things to comment on include the following, but you're not restricted to these:

- Was the lab too long or too short?
- Were the readings appropriate for the lab?
- Was it clear (at least after you completed the lab) what the points of the exercises were?

- Are there additions or changes you think would make the lab better?

Type your answer here, replacing this text.

End of lab 2-2

To double-check your work, the cell below will rerun all of the autograder tests.

[ ]:
```
grader.check_all()
```