Natural Language Understanding, Generation, and Machine Translation (2018–2019)

Clara Vania and Adam Lopez
School of Informatics, University of Edinburgh

Credits: earlier versions developed by Mirella Lapata, Philip Gorinski, Frank Keller, and Richard Socher

NLU+ Coursework 1: Recurrent Neural Networks

This assignment is due 7th February 2019, 4pm.

Deadline for choosing partners: Thursday, the 31st of January at 4pm.

Executive Summary. This first part of this coursework will walk you through the implementation of a few critical parts of a recurrent neural network and the backpropagation algorithm, which are foundational components of the modern NLP toolkit. You will not need to derive anything mathematically—the maths will be given to you. This is similar to a lot of practical work in research and industry—it is *much* more common to implement and test an existing model from a specification than to derive a new one from first principles. In fact, what we ask you to implement is relatively low level, compared to what you can do with modern machine learning libraries. Nevertheless, the implementation-from-specification pattern is similar, and we hope that revealing more of the mathematical details will help demystify the underlying algorithms for you, and give you some degree of comfort with them. If you are very interested in deriving algorithms like this, we encourage you to dig further into the mathematics, but nothing in the coursework requires this.

The next parts of the coursework ask you to experiment with training regimes, and to adapt the model to an interesting psycholinguistic task that tests the model's behaviour on a phenomenon that humans process effortlessly—number agreement between subject and predicate in English. This will expose you to some contemporary cognitive science research that attempts to relate human and machine learning. We recommend that you read the accompanying research paper from which the coursework dataset orginates (Linzen *et al.*, 2016). Keen students are encouraged to read the followup paper by Gulordava *et al.* (2018) for further ideas.

Finally, there is an open-ended final question for especially keen students. You do not need to answer this question to receive a good mark, and should only attempt it if you are confident in your solutions to the rest of the coursework and you have sufficient time remaining.

Submission Deadline and Pacing. The coursework is due on Thursday, the 7th of February at 4pm, i.e. right before the Thursday lecture in week 4. If you want to work with a partner, you have an earlier deadline to relate this to us, on Thursday, the 31st of January at 4pm i.e. one week before the coursework due date.

There are five questions. Simply answering the first four will earn you a good mark, and they can be easily done in the available time. This is an empirical observation from previous offerings of this course, which included a similar (though not identical!) coursework. The optional open-ended part of the coursework is Question 5. If you want to attempt it, you will need to finish the first four questions several days early. This is a more ambitious schedule, so plan accordingly.

Pair work policy. You are *strongly* encouraged to work with a partner on this assignment. When you work with another person, you learn more, because you need to explain things to each other as you pool your collective expertise to solve problems. Explaining something to another person helps you debug your own thinking, and their questions help you overcome your own blind spots—something you cannot do on your own by staring at maths, code, or data. For this reason, it is best to seek partners with complementary skills to your own. You may not work in teams of three or more.

As a practical matter, students who work in pairs are likely to receive marks earlier, because solo submissions require additional marking hours, and the course enrollment is much higher than projected.

If you work with a partner, only one of you should submit your completed work. But I need to know that the submission represents the work of two people, and I need to know that reliably in advance in order to estimate marking hours. So, **you must do the following to ensure that both of you receive credit:**

1. Create a text file containing two lines. Each line should contain the UUN and name of one partner. So, your file should look something like this:

   ```
   s1234567 Student One
   s7654321 Student Two
   ```

2. Submit the file on DICE using this command:

   ```
   submit nlu+ cw1 partners.txt
   ```

You must do this by **Thursday, the 31st of January at 4pm**. You may not change partners after you have done this, so take this commitment seriously. If I do not receive a submission from your or your partner by the deadline, I will assume that you are working alone. Either way, I will confirm your choice with you shortly after the deadline to avoid later confusion. **I advise you to choose your partner now and get to work.** Piazza offers a feature that allows you to search for a partner, and you are welcome to use this. I will not be in a position to assign partners for you, but by now you should know some of your classmates from ANLP or other courses.

Submission. Your solution should be delivered in two parts and uploaded to Blackboard Learn. **Do not include your name or your partner's name in either the code or the writeup.** The coursework will be marked anonymously since this has been empirically shown to reduce bias. (I anonymize the filenames before marking.)

For your writeup:

- Write up your answers in a file titled `<UUN>.pdf`. For example, if your UUN is `S123456`, your corresponding PDF should be named `S123456.pdf`.
- The answers should be clearly numbered and can contain text, diagrams, graphs, formulas, as appropriate. Do not repeat the question text. If you are not comfortable with writing math on Latex/Word you are allowed to include scanned handwritten answers in your submitted pdf. You will lose marks if your handwritten answers are illegible.
- On Blackboard Learn, select the Turnitin Assignment "Coursework 1 ANSWERS". Upload your `<UUN>.pdf` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `B123456`.
- Please make sure you have submitted the right file. We cannot make concessions for students who turn in incomplete or incorrect files by accident.

For your code and parameter files:

- Compress your code for `rnn.py` as well as your saved parameters `rnn.U.npy`, `rnn.V.npy`, and `rnn.W.npy` into a ZIP file named `<UUN>.zip`. For example, if your UUN is `S123456`, your corresponding ZIP should be named `S123456.zip`.
- On Blackboard Learn, select the Turnitin Assignment "Coursework 1 CODE". Upload your `<UUN>.zip` to this assignment, and use the submission title `<UUN>`. So, for above example, you should enter the submission title `S123456`.

Good Scholarly Practice Please remember the University requirement as regards all assessed work for credit. Details and advice about this can be found at:

`http://web.inf.ed.ac.uk/infweb/admin/policies/academic-misconduct`

and links from there. Note that you are required to take reasonable measures to protect your assessed work from unauthorised access. For example, if you put your work in a public repository then you must restrict access only to yourself and your partner. For your writeup, and particularly on the final question, you should pay close attention to the guidance on plagiarism. Your instructor is **very good** at detecting plagiarism that even Turnitin can't spot, and in one year reported 22 students for plagiarism found by hand. In short: the litmus test for plagiarism is not the Turnitin check—that is simply an automated assistant. If you have borrowed or lightly edited someone else's words, you have plagiarised. Write your report in your own words. We will not mark for eloquence—as long as we can clearly understand what you did, that is fine.

Assignment Data The necessary files for this assignment are available on the NLU course page.

Python Virtual Environment For this assignment you will be using Python along with a few open-source packages. These packages cannot be installed directly, so you will have to create a virtual environment. We are using virtual environments to make the installation of packages and retention of correct versions as simple as possible. For this assignment we are going to use Miniconda + Python 3.5.

The instructions below are for DICE. You are free to use your own machine, but we cannot offer support for non-DICE machines.

Open a terminal on a DICE machine and follow these instructions. We are expecting you to enter these commands in one-by-one. Waiting for each command to complete will help catch any unexpected warnings and errors.

Note: You can skip step (1) and (2) if you already have Miniconda installed in your machine.

1. Open a terminal on DICE machine and type the following command:
   ```
   $> wget https://repo.continuum.io/miniconda/Miniconda3-latest-Linux-x86_64.sh
   $> bash Miniconda3-latest-Linux-x86_64.sh
   ```
2. Restart your terminal.
3. Create conda virtual environment:
   ```
   $> conda create -n nlu python=3.5
   ```

4. Activate conda environment:
   ```
   $> source activate nlu
   ```

5. Install packages that we need:
   ```
   $> conda install numpy gensim pandas
   ```

You should now have all the required packages installed. You only need to create the virtual environment and perform the package installations (step 1-5) **once**. However, make sure you activate your virtual environment (step 4) **every time** you open a new terminal to work on your NLU assignment. Remember to use the `source deactivate` command to disable the virtual environment when you don't need it.

Terminology/definitions Most neural network components, as well as their architecture and functionality, can be described using *matrix* and *vector* mathematical operations. Matrix and vector notation in the literature is inconsistent, so for this assignment we will use the following conventions:

(1) *Matrices* are assigned bold capital letters, e.g., $\mathbf{U}$, $\mathbf{V}$, $\mathbf{W}$.

(2) *Vectors* are written in bold lower-cased letters, e.g., $\mathbf{x}$, $\mathbf{s}$, $\mathbf{net}_{in}$, $\mathbf{net}_{out}$.

(3) For a matrix $\mathbf{M}$ and a vector $\mathbf{v}$, $\mathbf{Mv}$ represents their *matrix-vector (dot) product.*[1]

(4) For vectors $\mathbf{v}$ and $\mathbf{w}$ of equal length n, $\mathbf{v} \circ \mathbf{w}$ represents their *element-wise product*:

$$\mathbf{v} \circ \mathbf{w} = [\mathrm{v}_0\mathrm{w}_0, \mathrm{v}_1\mathrm{w}_1, \ldots, \mathrm{v}_n\mathrm{w}_n]$$

Similarly, $\mathbf{v} + \mathbf{w}$ and $\mathbf{v} - \mathbf{w}$ express element-wise addition and subtraction.

(5) For vectors $\mathbf{v}$ and $\mathbf{w}$, $\mathbf{v} \otimes \mathbf{w}$ represents their *outer product.*[2] Cross product

(6) In recurrent neural networks, we process a *sequence* (or *time*), where each component is in a different state depending on the position in the sequence (or time step). We use the notation $\mathbf{M}^{(t)}$, $\mathbf{v}^{(t)}$ to refer to matrices and vectors at time step t.

Provided code and use of NumPy We provide the template file `rnn.py` which you must use to write your code. We also provide an additional module, `rnnmath.py`, which consists of helper functions you can use. Finally, we provide `test.py`, which performs a very basic test of your code. Please familiarize yourself with the provided code and make sure you **don't** change the provided function signatures.

[1] https://en.wikipedia.org/wiki/Matrix_multiplication
[2] https://en.wikipedia.org/wiki/Outer_product

Throughout this assignment, you are required to use NumPy methods for all matrix/vector operations. **Do not try to implement matrix/vector functionality on your own.** If you need help with NumPy, please refer to its documentation[3] and look for answers on Google before asking on Piazza.

Introduction

For this assignment, you are asked to implement some basic functionality of a Recurrent Neural Network (RNN) for Language Modeling (LM). Given a word sequence $w_1, w_2, \ldots, w_t$, a language model predicts the next word w_{t+1} by modeling:

$$P(w_{t+1} \mid w_1, \ldots, w_t).$$

Below, you will be introduced to the main elements of a simple RNN for LM, based on the model proposed by Mikolov *et al.* (2010). In Question 2, you will implement its core word prediction functionality and its training by implementing a loss function and the model's gradient accumulation through backpropagation. In Question 3, you will train and fine-tune your language model on actual data and use it to generate sentences. In Question 4, you will adapt your language model to a new task.

Recurrent Neural Networks

A recurrent neural network for language modeling uses feedback information in the hidden layer to model the "history" $w_1, w_2, \ldots, w_t$ in order to predict w_{t+1}. Formally, at each time step, the model needs to compute:

$$\mathbf{s}^{(t)} = f\left(\mathbf{net}_{in}^{(t)}\right) \tag{1}$$

$$\mathbf{net}_{in}^{(t)} = \mathbf{Vx}^{(t)} + \mathbf{Us}^{(t-1)} \tag{2}$$

$$\hat{\mathbf{y}}^{(t)} = g\left(\mathbf{net}_{out}^{(t)}\right) \tag{3}$$

$$\mathbf{net}_{out}^{(t)} = \mathbf{Ws}^{(t)} \tag{4}$$

where $f()$ and $g()$ are the *sigmoid* and *softmax* activation functions respectively, $\mathbf{x}^{(t)}$ is the one-hot vector representing the vocabulary index of the word w_t, $\mathbf{net}_{in}^{(t)}$ and $\mathbf{net}_{out}^{(t)}$

[3] http://docs.scipy.org/doc/numpy/reference/index.html

are the activations for the hidden and output layers, and $\mathbf{s}^{(t)}$ and $\hat{\mathbf{y}}^{(t)}$ are the corresponding hidden and output vectors produced after applying the *sigmoid* and *softmax* nonlinearities.

For a given input $[w_1, w_2, \ldots, w_t]$, the probability of the next word at time step $t+1$ can be read from the output vector $\hat{\mathbf{y}}^{(t)}$:

$$P(w_{t+1} = j \mid w_t, \ldots, w_1) = \hat{y}_j^{(t)} \tag{5}$$

The parameters to be learned are:

$$\mathbf{U} \in \mathbb{R}^{D_h \times D_h} \quad \mathbf{V} \in \mathbb{R}^{D_h \times |V|} \quad \mathbf{W} \in \mathbb{R}^{|V| \times D_h} \tag{6}$$

where $\mathbf{U}$ is the matrix for the recurrent hidden layer, $\mathbf{V}$ is the input word representation matrix, $\mathbf{W}$ is the output word representation matrix, and D_h is the dimensionality of the hidden layer.

Question 1: Training RNNs [30 marks]

For a more graphical explanation of what you are intended to do, see the supplemental notes at the end of this document.

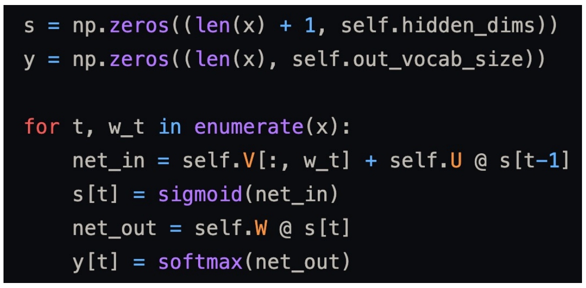
When training RNNs, we need to propagate the errors observed at the output layer $\hat{\mathbf{y}}$ back through the network, and adjust the weight matrices $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ to minimize the observed loss w.r.t. a desired output. There are several loss functions suitable for use in RNNs. In RNN language models, an effective loss function is the cross-entropy loss:

$$J^{(t)}(\theta) = -\sum_{j=1}^{|V|} d_j^{(t)} \log \hat{y}_j^{(t)} \tag{7}$$

where $\mathbf{d}^{(t)} = [d_1^{(t)}, d_2^{(t)}, \ldots, d_{|V|}^{(t)}]$ is the one-hot vector representing the vocabulary index of desired output word at time t. In order to evaluate the model's performance, we average the cross-entropy loss across all steps in a sentence and across all sentences in the dataset.

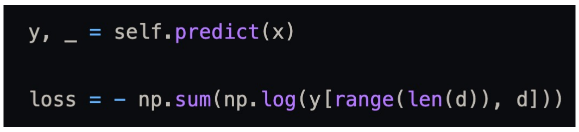
(a) In the file `rnn.py`, implement the method `predict` of the `RNN` class. The method is used for *forward prediction* in your RNN and takes as input a sentence as a list of word indices $[w_1, \cdots, w_n]$. The return values are the matrices produced by concatenating hidden vectors $\mathbf{s}^{(t)}$ and output vectors $\hat{\mathbf{y}}^{(t)}$ for $t = 1, 2, \ldots, n$.[4] **[5 marks]**

[4]See the provided documentation on `rnn.py` for more details on the functions you need to implement.



```
s = np.zeros((len(x) + 1, self.hidden_dims))
y = np.zeros((len(x), self.out_vocab_size))

for t, w_t in enumerate(x):
    net_in = self.V[:, w_t] + self.U @ s[t-1]
    s[t] = sigmoid(net_in)
    net_out = self.W @ s[t]
    y[t] = softmax(net_out)
```



```python
y, _ = self.predict(x)

loss = - np.sum(np.log(y[range(len(d)), d]))
```

(b) In `rnn.py`, implement the methods `compute_loss` and `compute_mean_loss`. Given a sequence of input words $w = [w_1, \dots, w_n]$ and a sequence of desired output words $d = [d_1, \dots, d_n]$, `compute_loss` should return the total loss produced by the model's predictions for the sentence. The `compute_mean_loss` should compute the average loss over a corpus of input sentences. It should average across all words in all sentences of the given corpus. **[5 marks]**

Optimizing the loss using backpropagation means we have to calculate the update values Δ w.r.t. the gradients of our loss function for the observed errors. For the output layer weights, at time step t we accumulate the matrix $\mathbf{W}$ updates using:

$$\Delta\mathbf{W} = \eta \sum_{p=1}^{n} \delta_{out,p}^{(t)} \otimes \mathbf{s}_p^{(t)} \quad \text{Bp inside activation} \tag{8}$$

$$\delta_{out,p}^{(t)} = (\mathbf{d}_p^{(t)} - \hat{\mathbf{y}}_p^{(t)}) \circ g'(\mathbf{net}_{out,p}^{(t)}) \quad \text{Bp last activation} \tag{9}$$

where η is the learning rate and p indicates the index of the current training pattern (sentence). We then further propagate the error observed at the output back to $\mathbf{V}$ with:

$$\Delta\mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{x}_p^{(t)} \tag{10}$$

$$\delta_{in,p}^{(t)} = \mathbf{W}^T \delta_{out,p}^{(t)} \circ f'(\mathbf{net}_{in,p}^{(t)}) \tag{11}$$

The derivatives of the softmax and sigmoid functions are respectively given as[5]:

$$g'(\mathbf{net}_{out,p}^{(t)}) = \vec{1} \tag{12}$$

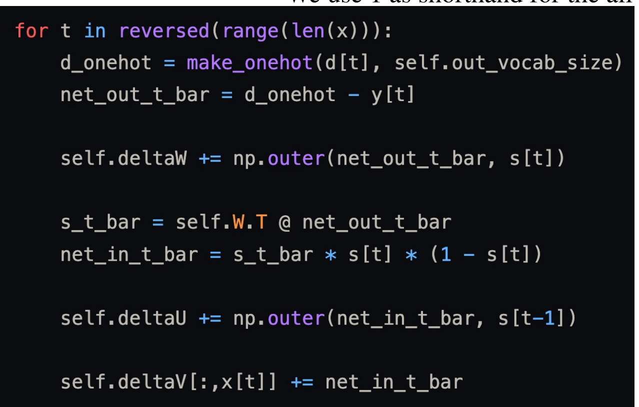
$$f'(\mathbf{net}_{in,p}^{(t)}) = \mathbf{s}_p^{(t)} \circ (\vec{1} - \mathbf{s}_p^{(t)}) \tag{13}$$

Finally, in order to update the recurrent weights $\mathbf{U}$, we need to look back one step in time:

$$\Delta\mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t)} \otimes \mathbf{s}_p^{(t-1)} \tag{14}$$

(c) In `rnn.py`, implement the method `acc_deltas` that accumulates the weight updates for $\mathbf{U}$, $\mathbf{V}$ and $\mathbf{W}$ for a simple backpropagation through the RNN, where we only look back one step in time as described above. **[10 marks]**

[5] We use $\vec{1}$ as shorthand for the all-ones vector of appropriate length.



```python
for t in reversed(range(len(x))):
    d_onehot = make_onehot(d[t], self.out_vocab_size)
    net_out_t_bar = d_onehot - y[t]

    self.deltaW += np.outer(net_out_t_bar, s[t])

    s_t_bar = self.W.T @ net_out_t_bar
    net_in_t_bar = s_t_bar * s[t] * (1 - s[t])

    self.deltaU += np.outer(net_in_t_bar, s[t-1])

    self.deltaV[:,x[t]] += net_in_t_bar
```

Look back one step

Now we have implemented simple backpropagation (BP) for recurrent networks—that is, RNNs that just look at the previous hidden layer when accumulating $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$. An extension to standard BP is backpropagation through time (BPTT), which takes into account the previous τ time steps during backpropagation. At time t, the updates $\Delta\mathbf{W}$ can be derived as before. For $\Delta\mathbf{U}$ and $\Delta\mathbf{V}$, we additionally recursively update at times $(t-1)$, $(t-2)$ $\cdots(t-\tau)$:

$$\Delta\mathbf{V} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{x}_p^{(t-\tau)} \tag{15}$$

$$\Delta\mathbf{U} = \eta \sum_{p=1}^{n} \delta_{in,p}^{(t-\tau)} \otimes \mathbf{s}_p^{(t-\tau-1)} \tag{16}$$

$$\delta_{in,p}^{(t-\tau)} = \mathbf{U}^T \delta_{in,p}^{(t-\tau+1)} \circ f'(\mathbf{net}_{in,p}^{(t-\tau)}) \tag{17}$$

(d) Implement the method `acc_deltas_bptt` that accumulates the weight updates for **U**, **V** and **W** using backpropagation through time for τ time steps. **[10 marks]**

New from 2017 cw:

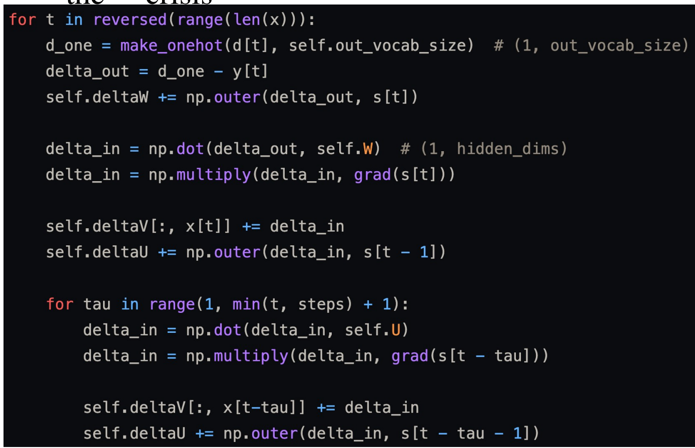
There's one last thing you'll need to do before your models will train: you'll need to complete the implementation of the `train-lm` mode in the `__main__` method of the code. For this, you will minimally need to instantiate the RNN class, call the `train` method with the appropriate arguments, and save the resulting matrices. You may find it useful for your own understanding to log different aspects of the training process here.

You do not need to report anything in your writeup for Question 1. It will be evaluated solely on the basis of your code.

Question 2: Language Modeling [20 marks]

By now you should have everything in place to train a full Recurrent Neural Network using backpropagation through time. In the following questions, we will use the training and development data provided in `wiki-train.txt` and `wiki-dev.txt`. The training data consists of sentences from the parsed English Wikipedia corpus from Linzen *et al.* (2016), and each input/output pair x, d is of the form $[w_1, \cdots w_n]$ / $[w_2, \cdots w_{n+1}]$ that is, the desired output is always the next word of the current input:

time index	t=1	t=2	t=3	t=4
input:	Banks	struggled	with	the
output:	struggled	with	the	crisis


```
for t in reversed(range(len(x))):
    d_one = make_onehot(d[t], self.out_vocab_size)  # (1, out_vocab_size)
    delta_out = d_one - y[t]
    self.deltaW += np.outer(delta_out, s[t])

    delta_in = np.dot(delta_out, self.W)  # (1, hidden_dims)
    delta_in = np.multiply(delta_in, grad(s[t]))

    self.deltaV[:, x[t]] += delta_in
    self.deltaU += np.outer(delta_in, s[t - 1])

    for tau in range(1, min(t, steps) + 1):
        delta_in = np.dot(delta_in, self.U)
        delta_in = np.multiply(delta_in, grad(s[t - tau]))

        self.deltaV[:, x[t-tau]] += delta_in
        self.deltaU += np.outer(delta_in, s[t - tau - 1])
```

The `utils.py` module provides functions to read the Wikipedia data, and the `__main__` method of the `rnn.py` module provides some starter code for training your models. Use mode `train-lm` to train your language model.

the same as 2017 cw

Only format changed

(a) Perform parameter tuning using a subset of the training and development sets. You must use a fixed vocabulary of size 2,000, and consider all combinations of the following parameter settings.

learning rate: 0.5, 0.1, or 0.05 number of hidden units: 25 or 50
number of steps to look back in truncated backpropagation: 0, 2, or 5

The mode `train-lm` in `rnn.py` allows for more parameters, which you are free to explore. You should tune your model to maximize generalization performance (minimize cross-entropy loss) on the dev set. For these experiments, use the first 1,000 sentences of both the training and development sets and train for 10 epochs.[6] Report your findings **and interpret them**. **[10 marks]**

(b) Using your best parameter settings found in (a), train an RNN on a much larger training set. Use a fixed vocabulary size of 2000, train on 25,000 sentences, and, as before, use the first 1,000 development sentences to evaluate the model's performance during training. When your model is trained[7], evaluate it on the full dev set and report the mean loss, as well as both the adjusted and unadjusted perplexity your model achieves[8]. Save your final learned matrices **U**, **V** and **W** as files `rnn.U.npy`, `rnn.V.npy` and `rnn.W.npy`, respectively. **[10 marks]**

Question 3: Predicting Subject-Verb Agreement [15 marks]

The form of an English third-person present tense verb depends on whether the **head** of the syntactic subject is plural or singular. For example, native English speakers strongly prefer sentences (i) and (iv) below, and regard (ii) and (iii) as ungrammatical, as indicated by the *:

i) The **key** <u>is</u> on the table.

ii) *The **key** <u>are</u> on the table.

iii) *The **keys** <u>is</u> on the table.

[6]Note that training models might take some time. For example, a sweep of the parameters settings described above should take roughly 2 hours on a student lab DICE machine. Please avoid using `student.compute` to train your models as run times will become very slow on a busy server.

[7]This should also take roughly 2 hours. Again, avoid using `student.compute`.

[8]Use your method `compute_mean_loss` to calculate loss on the development set, and the provided method `adjust_loss` to get adjusted/unadjusted perplexities for your models

iv) The **keys** are on the table.

This agreement tends to persist even when the head of the subject is separated from the verb by intervening words:

v) The **keys** to the cabinet are on the table.

Agreement rules like this occur in many languages, and are often more complex than in English. Our goal for this question will be to test (in a limited way) whether an RNN can learn them. For our first test, we will train a model predict agreement using direct supervision. That is, we will give our model the sequence of words preceding the verb, and we will ask it to predict whether the verb is singular (VBZ), or plural (VBP). Our training and test data will be in this form:

time index	t=1	t=2	t=3	t=4	t=5
input:	The	keys	to	the	cabinet
output:					VBP

Since the head of the subject may be arbitrarily far from the verb, this problem is a natural application of RNNs, which can encode the input sentence. But since the task is now binary classification, we must make some changes to the RNN. Instead of making predictions at **every** time step, we only make a prediction at the **final** time step.

(a) Implement new functions for weight updates (`acc_deltas_bptt_np`), loss fuction (`compute_loss_np`), and prediction accuracy (`compute_acc_np`) to reflect the structure of the number prediction problem. **[10 marks]**

You'll need to complete the implementation of the `train-np` mode in the `__main__` method of the code. Check your implementation by running the code:

```
$> python rnn.py train-np data_dir hdim lookback learning_rate
```

(b) Train your new model, explaining the parameters you used and how you chose them, and include the results in your report. **[5 marks]**

Question 4: Number Prediction with an RRNLM [4 marks]

Let's consider the problem from question 3 from a slightly different perspective, inspired by the Linzen *et al.* (2016) paper that provided our data. Human learners of language generally learn morphosyntactic features of language, like number agreement, with little to no direct supervision. Can a computational model like an RNN also do this? That is, can it learn agreement simply from the language data itself?

Let's return to the RNNLM that you implemented in Question 1 and trained in Question 2. Suppose once again that your input looks like this:

time index	t=1	t=2	t=3	t=4	t=5
Input $x =$	The	keys	to	the	cabinet

To use an RNN for the task in question 3, we simply say:

$$\text{Output} = \begin{cases} \text{VBZ if } P(\text{is} \mid x) > P(\text{are} \mid x) \\ \text{VBP otherwise} \end{cases}$$

Implement method `compare_num_pred` and evaluate your prediction accuracy:

```
$> python rnn.py predict-lm rnn.py data_dir rnn_dir
```

Include your result in your final report. **[4 marks]**

Question 5: Exploration [31 marks]

The final part of this question is open-ended, and it is intended primarily for students who want to deepen their knowledge, for the price of some additional work. **You are not required to attempt this part of the assignment**, and you will receive a good mark if you do a good job on only the first four questions. If you are not yet comfortable with the material in Questions 1–4, focus on those questions only.

I want you to ask your **own** question about the number prediction task and RNNs, and then attempt to answer it. Put another way: I want you to develop a simple scientific hypothesis and test it. **Your answer should be supported by empirical evidence**, and **you must analyze this evidence**, so you'll need to look at the data, count something, and interpret the results with respect to your stated question.

You are welcome (but absolutely not required) to implement another model, if doing so helps you answer a question. But this question is not a test your implementation skills, and you will not receive more marks simply for writing more code. A simple, well-posed analysis or experiment, clearly explained, is much better than an implementation of the fanciest architecture that you found while browsing the latest papers on arxiv.org. It is also better than making numbers go up without insight. This is not a kaggle competition.

This question is also not a test of the **amount** of work that you do: I realize that you have limited time for this part of the coursework, and I don't expect you to produce a novel result or even a novel question. I *do* expect you to ask a small question that you find interesting, even if it's a question you've seen elsewhere, or just something small that intrigued you about the data. It is perfectly ok if you present an interesting question and run a well-posed analysis whose results are inconclusive. That is how most science works.[9] In short, the brief for this question is for you to be curious and engage with the topic of the course. Your answer should tell us something interesting that you learned (or attempted to learn).

This question will be marked according to the descriptors for the common marking scheme.[10] Note that these criteria require "elements of personal insight / creativity / originality" for a mark above 70—that's why this question is worth 31 marks. For the same reason, it will be rare for answers to this question to receive more than 15 or 16 points (only 2% of your overall mark), so **I urge you not spend too much time on it**. Your answer should not be more than two pages, including figures. **[31 marks]**

References

Kristina Gulordava, Piotr Bojanowski, Edouard Grave, Tal Linzen, and Marco Baroni. Colorless green recurrent networks dream hierarchically. In *Proceedings of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2018.

Jiang Guo. Backpropagation Through Time. *Unpubl. ms., Harbin Institute of Technology*, 2013.

Tal Linzen, Emmanuel Dupoux, and Yoav Goldberg. Assessing the ability of LSTMs to learn syntax-sensitive dependencies. *Transactions of the Association for Computational Linguistics*, 4:521–535, 2016.

Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernockỳ, and Sanjeev Khudanpur. Recurrent neural network based language model. In *INTERSPEECH*, volume 2, page 3, 2010.

Acknowledgements

This assignment is based in parts on code and text kindly provided by Richard Socher, from the course on "Deep Learning for Natural Language Processing".[11]

[9] "Research is the process of going up alleys to see if they are blind."—Marston Bates

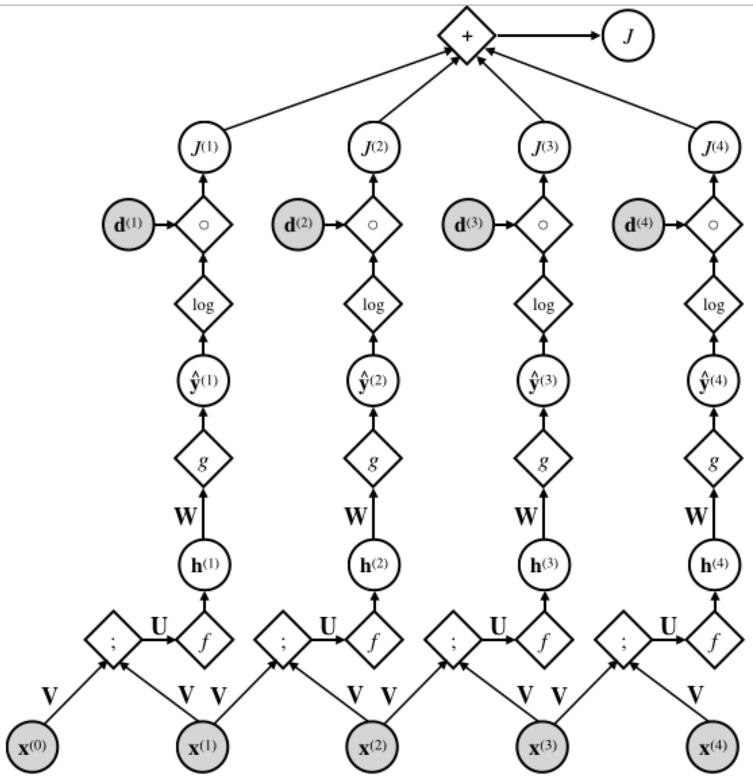
[10] `https://info.maths.ed.ac.uk/assets/files/Projects/college-criteria.pdf`

[11] http://cs224d.stanford.edu/

Backpropagation through time, Q1, and Q3

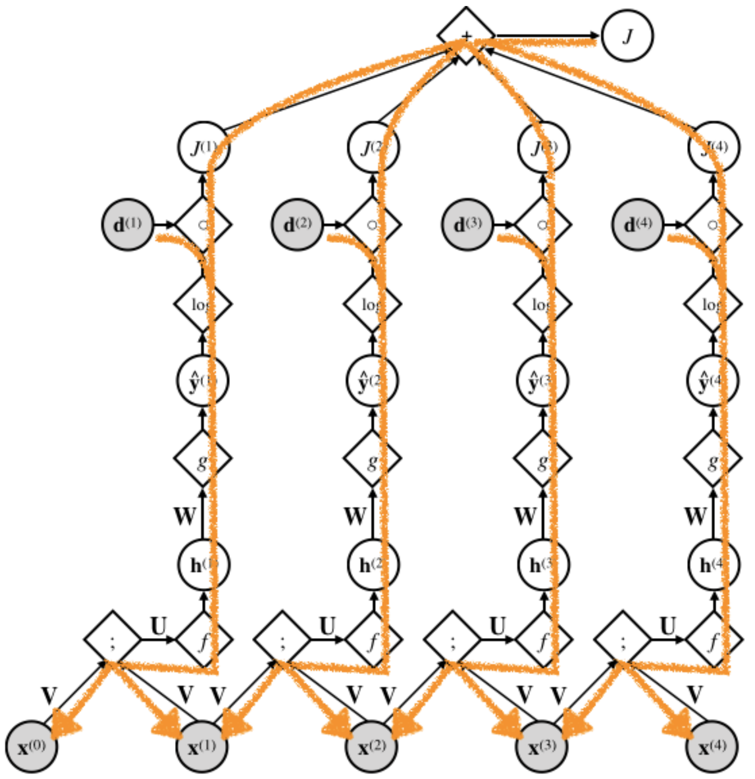
The purpose of this note is to clarify the relationship between backpropagation, backpropagation through time, and **truncated** backpropagation through time, which you are to implement for the coursework.

First, let's talk about backpropagation on a simple feedforward network---in other words, a classifier whose structure is the same for every problem instance. For concreteness, we'll use a feedforward trigram language model, which has almost the same structure as our RNN, except that instead of depending on all history words through a recurrent state, each prediction depends only on the two most recent words. It's illustrated below as a computation graph using the same notation as in your homework where possible. (graphically: shaded circles are observed values; circles are values in general; diamonds are functions; parameter matrices are edge labels, and these edges indicate multiplication by these matrices).

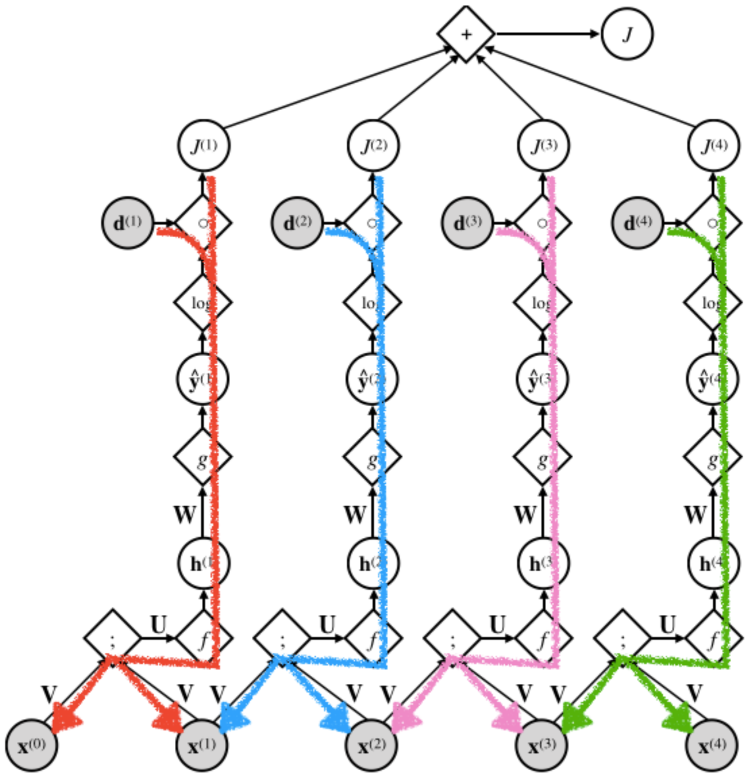


Notice that I've included that overall loss, J, because our goal is to set **U**, **V**, and **W** to minimize J---the entire computation is aimed at minimizing this single function. We do this numerically by taking the partial derivative of J w.r.t. each parameter matrix and then running some variant of a gradient descent algorithm. **The role of backpropagation is to compute the gradient.** For a really clear picture of how that works, I recommend reading the first few sections of Justin Domke's excellent notes on automatic differentiation: https://people.cs.umass.edu/~domke/courses/sml2011/08autodiff_nnets.pdf

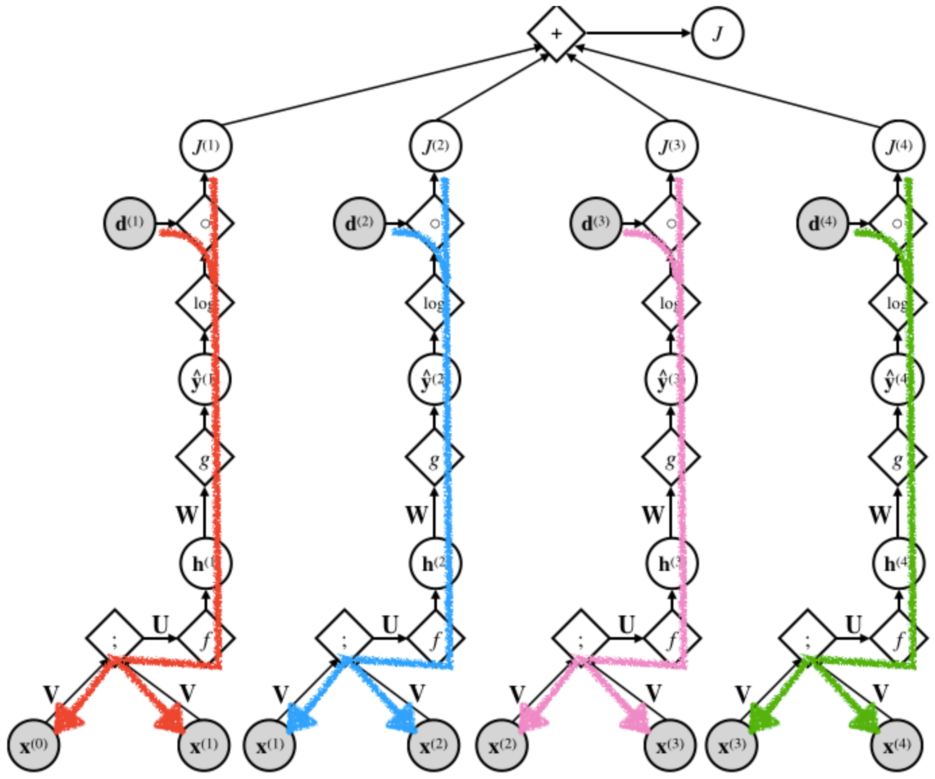
In this case, backpropagation from J (the output) recursively computes partial derivatives in reverse topological order on the computation graph. The partial derivative from backpropagation will also be affected by the **d** nodes. So, its structure looks like this:



One thing to notice here is the structure of the loss J: it's simply the sum of the losses incurred at each individual time step. Since the derivative of a sum of functions is simply the sum of the derivatives of those functions, that means that we can break the computation into four individual pieces. That looks like this:



Just so there's no confusion, notice that the individual losses share absolutely no substructure other than observed variables. If it helps to make that clear, you can also think of the computation like this:

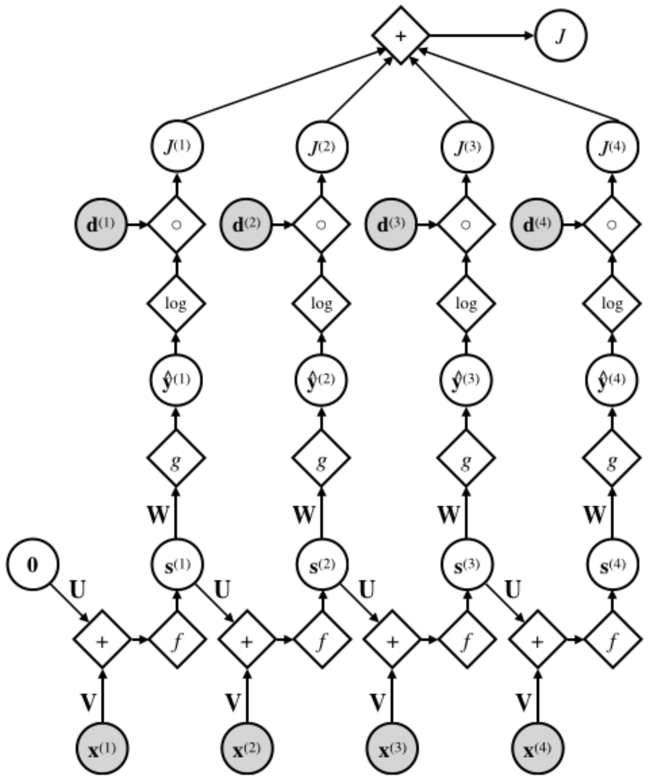


A couple of things are worth pointing out here:

* Your goal is still to compute partial derivatives w.r.t. J. But you can do that by computing partial derivatives w.r.t. the $J^{(i)}$ variables and then summing up as you go.

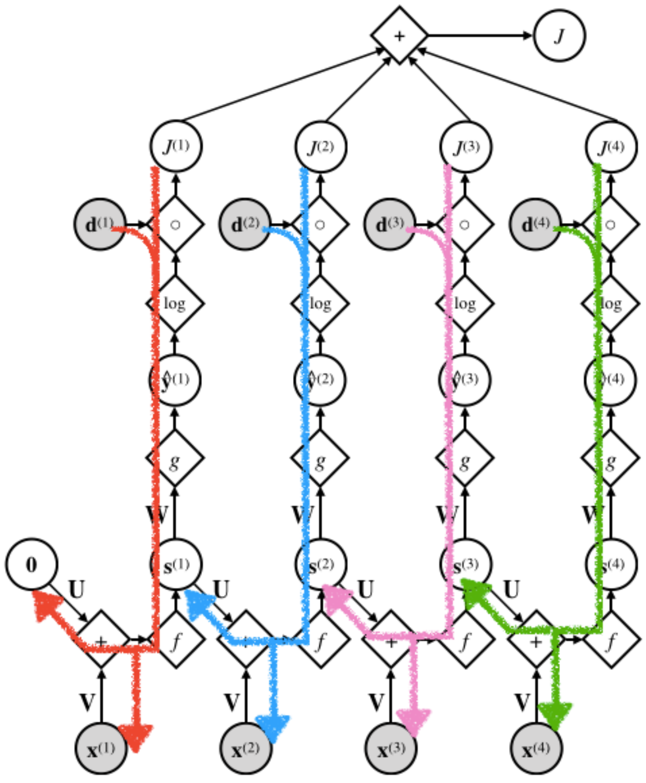
* The structure of these individual computations is always identical. That means that we can construct a small computation subgraph once and run backpropagation on it many times, simply replacing the observed values for each training instance. That makes it easy to implement and train the feedforward network.

The idea in Q1c is to simply port this simple computation to a RNN-LM. First, let's take a look at the structure of that model, using the same notation as in the coursework.

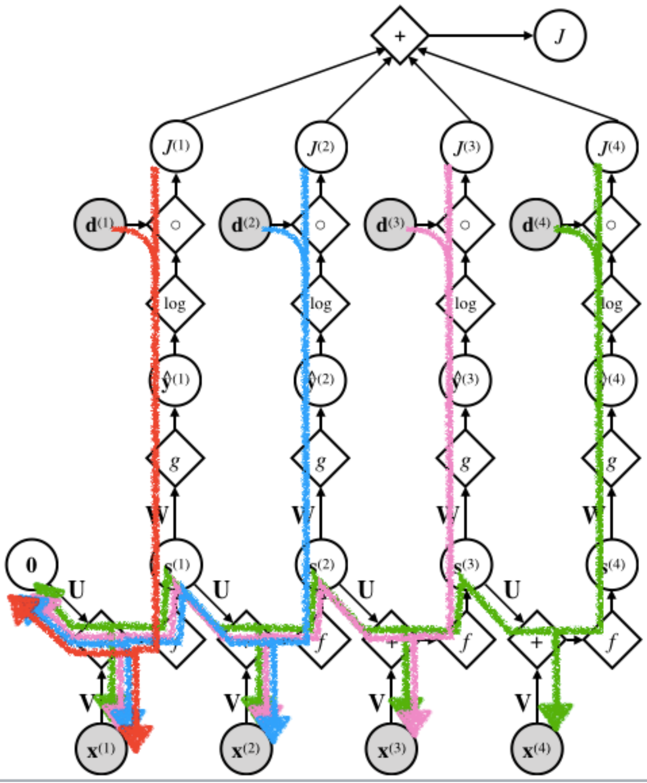


Notice in particular that the individual loss functions share substructure, specifically the **s** nodes. And this means that the indvidual loss functions are now **dynamic**: the function underlying a particular loss depends on the shape of the input. That is, the computation leading to $J^{(1)}$ has a different shape than the computation leading to $J^{(4)}$. A correct implementation of backpropagation would need to account for this, which should be quite clear from Domke's notes above.

Despite this, Q1c just asks you to implement backpropagation on a fixed subgraph that has a common shape for each of the indvidual losses. That is, it asks you to do this:



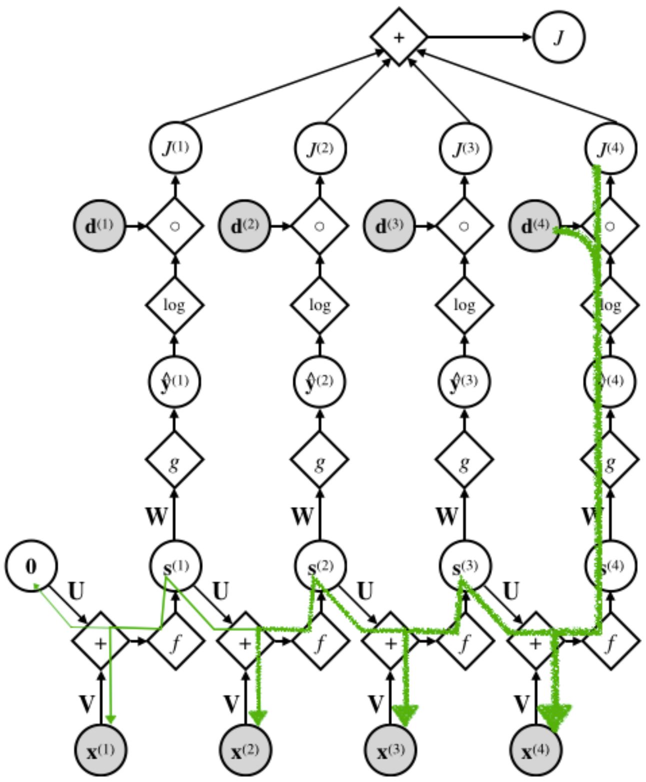
Let's stop and look at a few things about this. First, this is not correctly computing the partial derivative: the loss at t=4 does in fact depend on the input at t={1,...,4}, and we haven't backpropagated that far. So what we're getting here is an approximation. In fact, to compute the correct partial derivatives, we would need to do this:



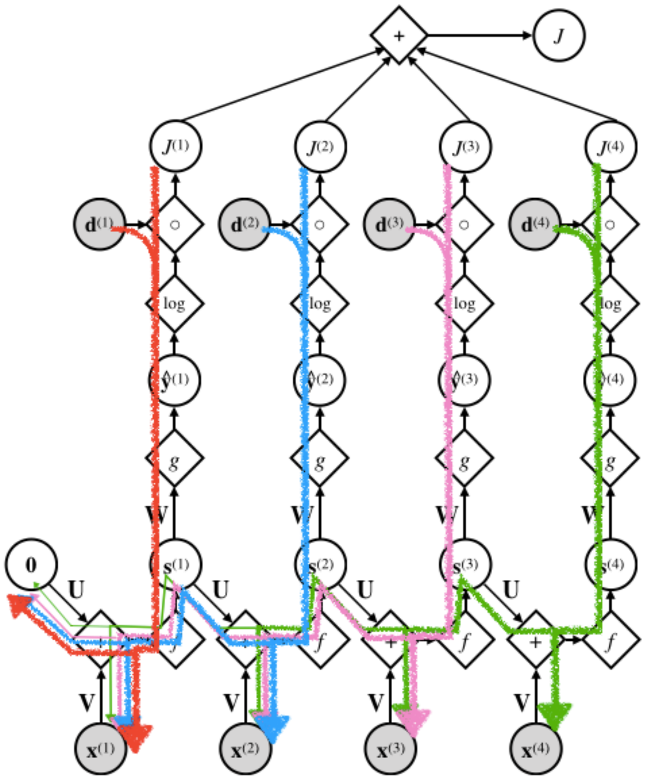
The structure illustrated above is what's often called **backpropagation through time**. But just so we're clear what that means: **it's just backpropagation** on a dynamic computation graph. It's this dynamic computation graph that's different, not the computation of derivatives. In this sense, "backpropgation" and "backpropagation through time" are really the same algorithm, and if any algorithm is really an outlier, it's the version that we ask you to compute in Q1c. That algorithm is properly called **truncated backpropagation through tim**e.

[Two minor parenthetical notes that you can safely skip:
1) There are are a bunch of different variants on truncated BPTT, but they all use roughly the same idea.
2) The structure above implies a quadratic algorithm, which might worry you. But this is just for illustration. You can in fact do this in linear time by propagating the gradients in strict reverse topological order---simply accumulate the partial gradients from each parent node, and only backpropagate once you have all of them. Memory use does grow linearly with the sequence though. See Domke's notes for more intuition about this.]

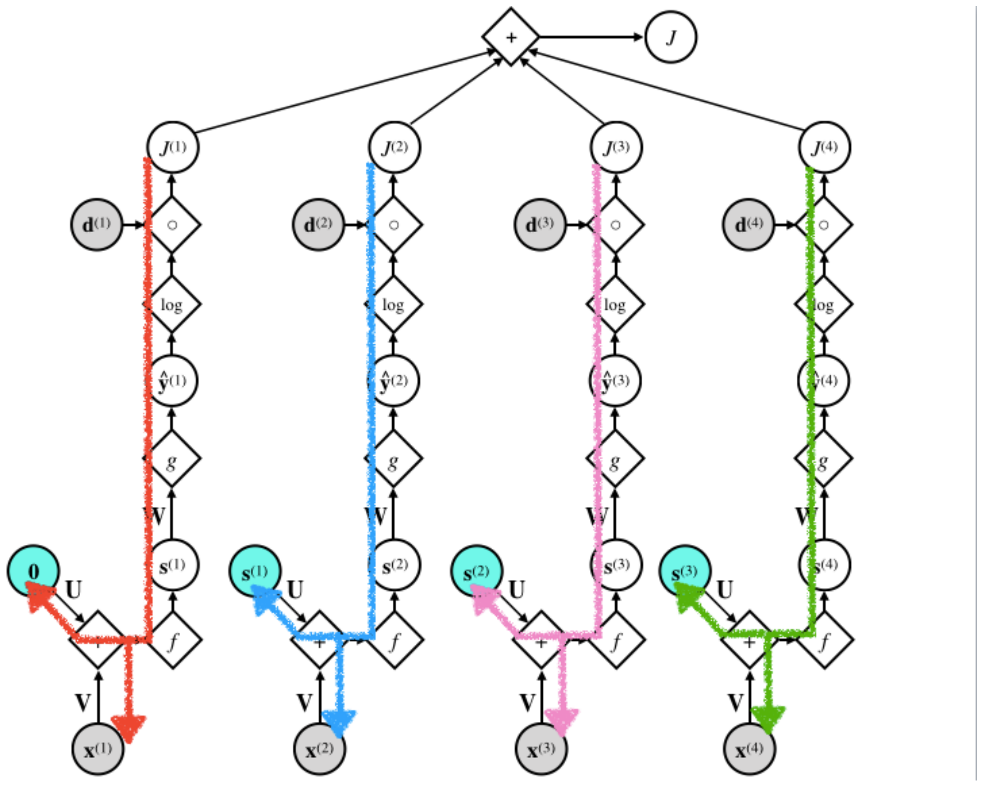
Of course, this isn't the full story. As we discussed in class, the partial derivatives from each individual loss diminish over time: the **vanishing gradient problem** is w.r.t. an indvidual loss.



So, the full BPTT computation behaves more like this:



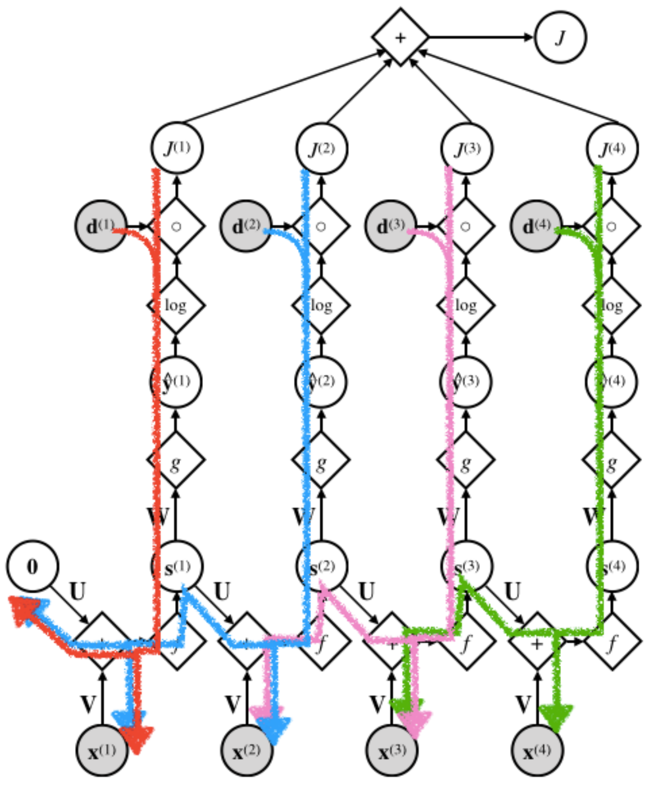
From this perspective, truncated BPTT almost like a reasonable approximation... except that now it might seem as if it has no way to capture long-distance dependencies, which is the whole point of using an RNN! But this is not quite true, and to see why, let's look at the individual computations in truncated BPTT, much as we did for the feedforward case above:



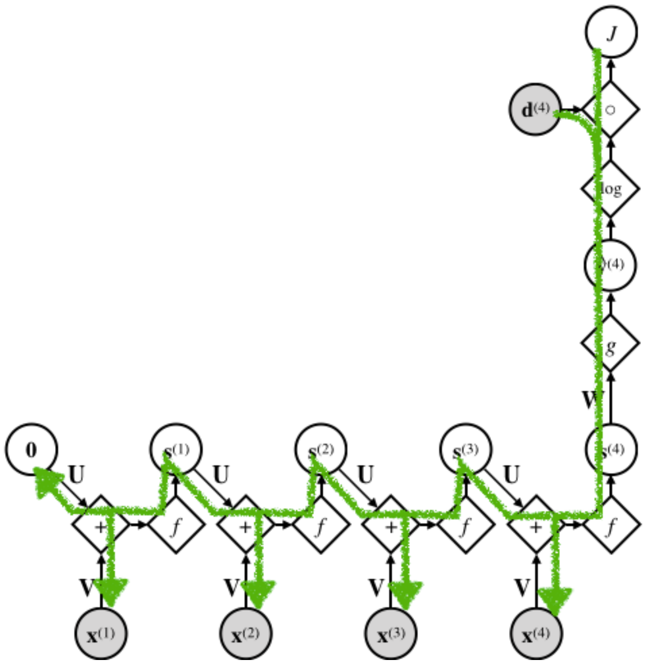
I've highlighted the RNN states here to make a point: allthough TBPTT treats these as if they were observed input variables, they aren't! They are, in fact, the result of first running a forward pass of the RNN. When you run TBTT on this computation, updates to the **U** matrix do reflect dynamic properties of the RNN that depend on the full history. It is true, however, that we cannot backpropagate signal explicitly between words with long-distance dependencies. To have any hope of capturing these, we need for **s** to learn to remember information that enters the network via matrix **V**, and for **V** to be well-trained based on local dependencies. If most dependencies are local, then this might happen.

Of course, whether that really happens is an empirical question, and it depends on the properties of your data and your problem. It is not magic and you should simply assume that it will happen.

Q1d is **also** asking you to implement truncated BPTT, but simply unrolling the network a bit further (this observation may give you a clue about how to simplify your code). For example, if you look back one step, you get this:



Now let's look at what we're asking you to do in Q3. In a single picture, we're just asking you to implement the **truncated** BPTT version of this:



Implementation-wise, if you suspect that this is very similar to your other solutions, you are probably on the right track. There are a lot of interesting questions that this new task raises, so we've given you some space to explore them in Q4.