Lab 2
Language Models

Prepared by:
Omar Samir

Lab Outcomes

- Implement Language Models from scratch
    - Bi-gram Language Model
    - RNN Language Model

- Applications of Language Models
    - Predicting randomly masked words
    - Generating new sentences

Language Models

A language model is a model that predicts some word ***w*** given some history of words ***h*** by computing the following formula

$$P(w|h)$$

By calculating this formula for all possible words, we can determine the most possible words.

Bi-gram Language Model

The bi-gram LM makes this approximation

$$P(w_n|w_{1:n-1}) \approx P(w_n|w_{n-1})$$

This probability can be calculated just by counting frequencies

$$P(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)}{C(w_{n-1})}$$

With add-one smoothed bigram our final equation will be

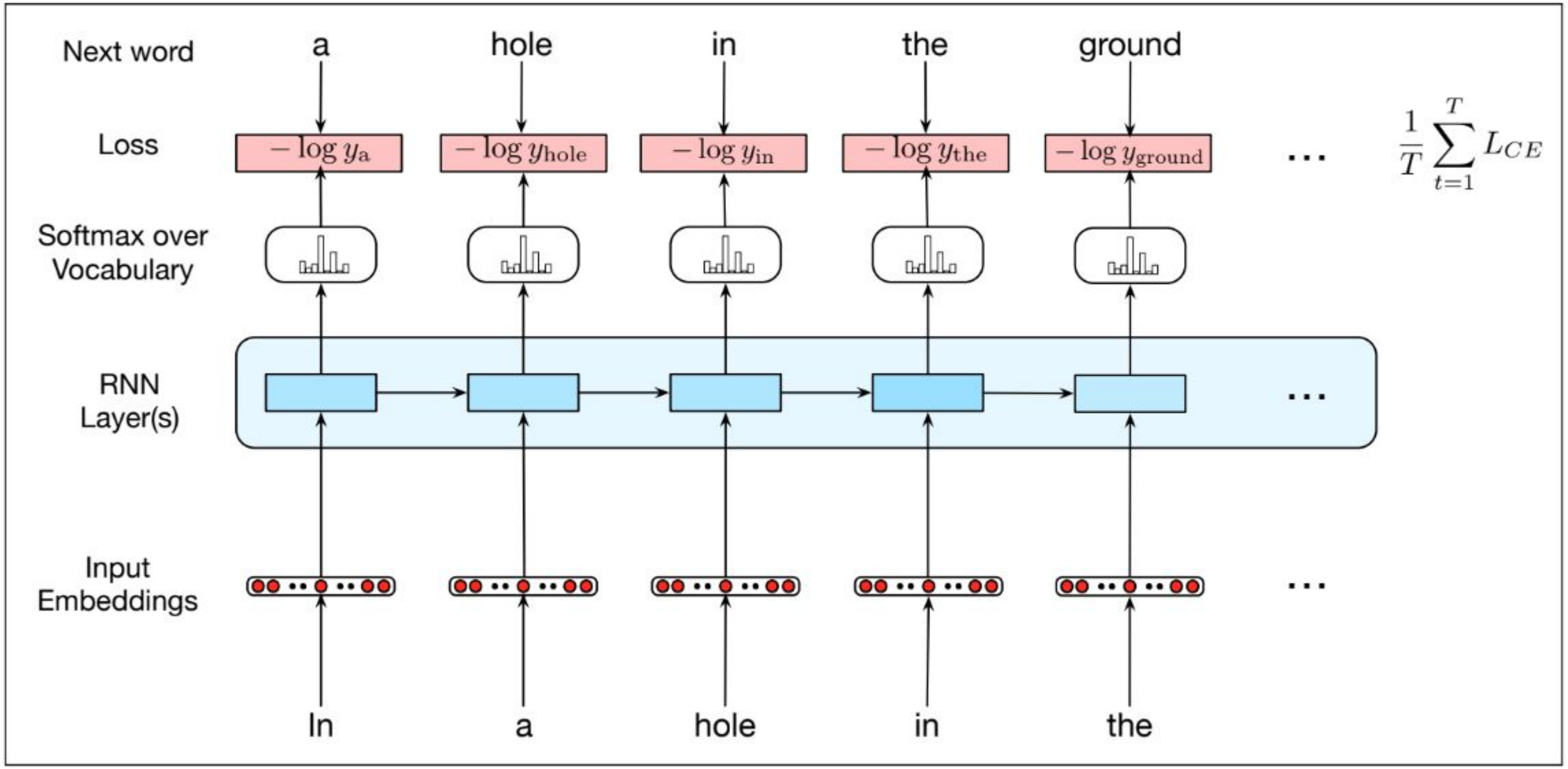
$$P^*_{\text{Laplace}}(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{\sum_w (C(w_{n-1}w)+1)} = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V}$$

Bi-gram Language Model

To implement it we need to do the following:

1. Create the bi-gram matrix of size |V| * |V| where |V| is the size of vocabulary
2. Loop over sentences to create all possible bi-grams
3. Use these counts to fill the matrix
4. To calculate the probability we can use the matrix we constructed with the add-one smoothed formula

RNN Language Model



RNN Language Models

- Since we need to work with matrices and vectors, we need a library that does the basic matrix operations.
- The library we will use is numpy which contains all matrix operations we need

```
>>> import numpy as np

>>> np.dot()

>>> np.sum()

>>> np.exp()

>>> x.T
```

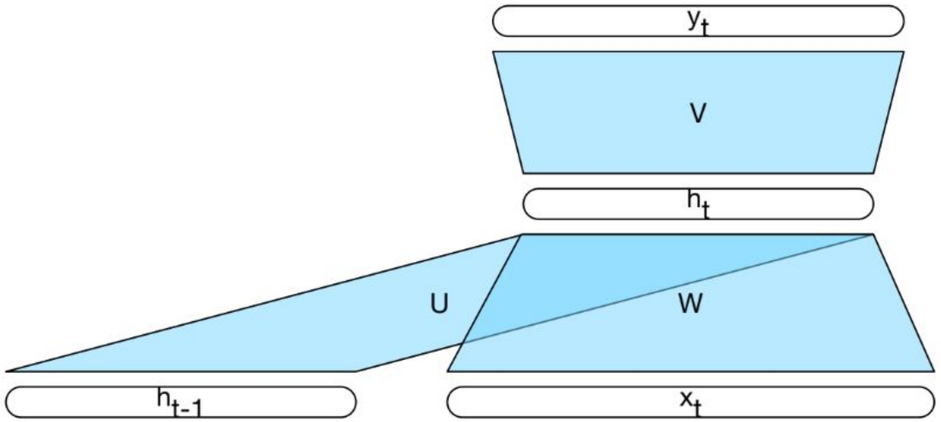
Input Embedding Layer

- This layer is used to convert the input token to an embedding vector of side D
- There are several options to get embeddings
  - One hot encoding (The simplest one)
  - Pre-trained word embedding (like word2vec)
  - Trainable embeddings

- In our case we will work with one hot encoding so the whole matrix will be of size |V| * |V| and it will be an identity matrix

RNN Language Model

- To initialize your RNN, you need to determine what are the weights
  - W or (W_x) of shape (D, H)
  - U or (W_h) of shape (H, H)
  - b (bias) of shape (H,)
  - V or (W_y) of shape (H, V)
  - b_y of shape (V,)
- Where
  - D is the embedding size
  - H is the hidden state size
  - V is the vocabulary size



RNN One Step Forward

At each time step, we need to calculate the new hidden state given the previous hidden state and the current input x. The equations are as follows:

$$a_t = W_h h_{t-1} + W_x x_t + b$$

$$h_t = \tanh(a_t)$$

In neural networks, we usually feed it with several sentences at the same time as we are working with matrices so we need to think how the sizes will be to make sure our operations are working correctly

RNN One Step Forward

$$a_t = W_h h_{t-1} + W_x x_t + b$$

$$h_t = \tanh(a_t)$$

The sizes are as follows:

- x_t of shape (N, D)
- W_x of shape (D, H)
- h_{t-1} of shape (N, H)
- W_h of shape (H, H)
- b of shape (H,)
- a_t of shape (N, H)
- h_t of shape (N, H)

- Where N is the batch size
- Think how to order these matrices for correct output
- Matrix multiplication is done using np.dot()

RNN Gradients Calculations

- In training process, our goal is to calculate the gradient of the total loss with respect to all model weights to do the optimization
- Since the path has many operations, we use the chain rule and calculate all partial gradients.
- Our final goal is to calculate the following gradients
  - $dLoss/dW_x$
  - $dLoss/dW_h$
  - $dLoss/db$
  - $dLoss/dW_y$
  - $dLoss/db_y$

RNN One Step Backward

$a_t = W_h h_{t-1} + W_x x_t + b$

$h_t = \tanh(a_t)$

Since we are now looking only in one step, we will assume that $dLoss/dh_t$ will be given

Now let’s work backward

1. $dh_t/da_t = 1 - \tanh^2(a_t) = 1 - h_t^2$
2. $da_t/dh_{t-1} = W_h^T$
3. $da_t/dW_h = h_{t-1}^T$
4. $da_t/dW_x = x_t^T$
5. $da_t/db = 1$

RNN One Step Backward

Now we can use chain rule as follows:

1. $dLoss/da_t = dLoss/dh_t * dh_t/da_t$
2. $dLoss/dh_{t-1} = dLoss/da_t * da_t/dh_{t-1}$
3. $dLoss/dW_h = dLoss/da_t * da_t/dW_h$
4. $dLoss/dW_x = dLoss/da_t * da_t/dW_x$
5. $dLoss/db = dLoss/da_t * da_t/db$

What about the sizes??

RNN One Step Backward

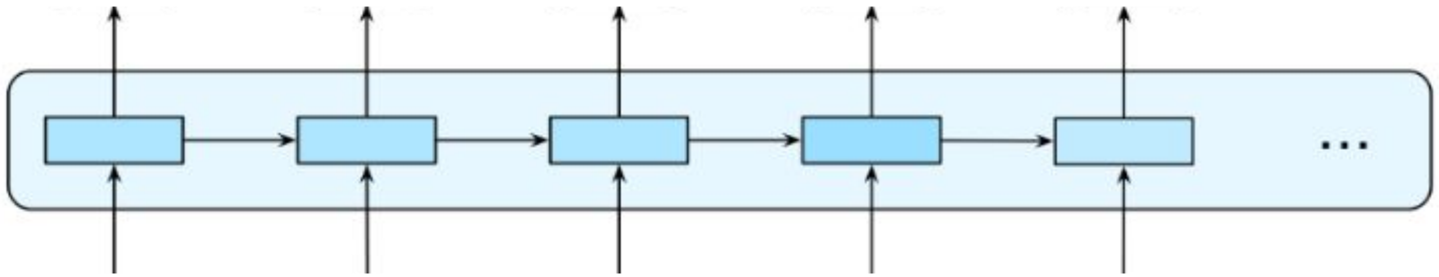
As you may notices, to calculate the derivatives, we need to cache values from the one step forward as follows:

1. x_t
2. W_h
3. h_{t-1}
4. h_t

RNN Full Forward Pass

To make the full pass we need to do the following:

1. Create a matrix h that will be filled with all hidden states with size (N, T, H)
    a. T is the size of sequence sentences
2. Loop over T and calculate one step forward with each input value
3. Cache all values in a list for the backward pass



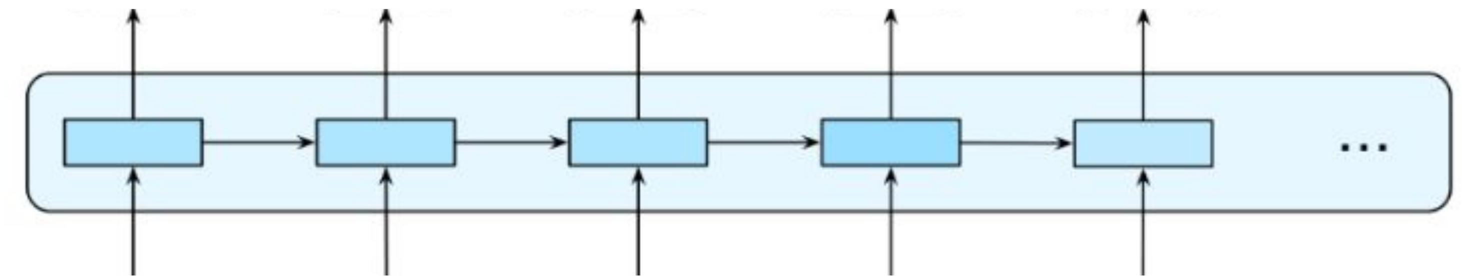
RNN Full Backward Pass

This backward pass will calculate gradients dW_x, dW_h, db

The function will assume that dLoss/dh of shape (N, T, H) will be given to it

1. Loop over T and call one_step_backward with the appropriate parameters
2. All gradients returned for Wx, Wh and b should be added
3. Return the final gradients

To pass the correct dh look at this figure



Hidden to Scores Forward

Till now we produced all hidden states having a matrix of shape (N, T, H)

For a Language model we need to produce scores for all words in our vocabulary

This can be done using another layer with the following equation

$$y = W_y h + b_y$$

Again what about sizes??

Hidden to Scores Backward

Here we assume we are given the gradient of loss with respect to scores (y) with the shape (N, T, V)

$y = W_y h + b_y$

The derivates will be as follows

1. $dLoss/dh = dLoss/dy * dy/dh = dLoss/dy * W_y^T \rightarrow$ This one will be given to full backward pass
2. $dLoss/dW_y = dLoss/dy * dy/dW_y = ((dLoss/dy)^T * h)^T$
3. $dLoss/db_y = dLoss/dy * dy/db_y = dLoss/dy$ (with summing)

Again what about sizes??

Softmax Loss

Now, since we have the scores we need to calculate the cross entropy loss.

This one will be implemented for you producing the loss as a scalar and dLoss/dy of shape (N, T, V)

Putting All Together

You will put all the previous components together in one big loss function that calculates the forward pass producing the total and all caches. Then you will do the full backward pass to produce all the gradients with respect to all model weights.

The function will return the following:

- Loss
- Gradients

The gradients of course will be used to update the model parameters using the gradient descent algorithm.

Language Models Applications

1. Prediction

Given a sequence of tokens you can use a language model to generate the probability distribution over your vocabulary then pick the token with the maximum probability

2. Generation

Since the language model produce a probability distribution over words, this distribution can be used for random sampling so generating different outcomes while preserving the LM probability distribution

The sampling can be done in numpy using np.random.choice