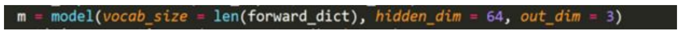
CS 4740: Intro to NLP
Project 3: Neural Models in NLP

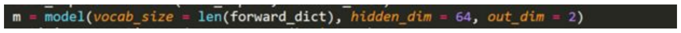
Jahanvi Kolte, Tian Ren, Xingyu Chen
jsk362@cornell.edu, tr349@cornell.edu, xc374@cornell.edu

I. Debugging

Error 1: The model output is defined as a vector of dimension of 3 in main.py.

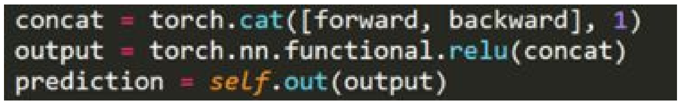

```
m = model(vocab_size = len(forward_dict), hidden_dim = 64, out_dim = 3)
```

Solution: Change the output dimension to 2 so that it fits this binary classification task where the possible label is either objective (label 0) or subjective (label 1).

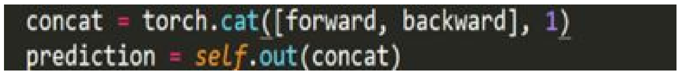

```
m = model(vocab_size = len(forward_dict), hidden_dim = 64, out_dim = 2)
```

This fixed the problem that the prediction now is made from 2 choices of labels instead of 3 labels.

Error 2: The relu activation function is set before the linear layer.

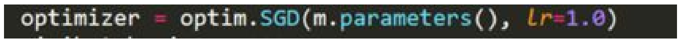

```
concat = torch.cat([forward, backward], 1)
output = torch.nn.functional.relu(concat)
prediction = self.out(output)
```

Solution:

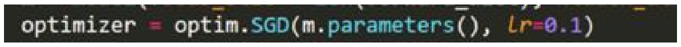

```
concat = torch.cat([forward, backward], 1)
prediction = self.out(concat)
```

The relu activation usually should be put after the linear layer. However, the linear layer is at the output and we are doing classification task, therefore instead of using relu we should use a softmax layer. But the 'CrossEntropy' loss function in Pytorch contains a softmax layer already thus now we don't need explicitly set any activation at the output layer. Thus, we only need to delete the relu activation to correct this bug. Moreover, the GRU unit already has non-linear activation function and concatenation will not change any number representation.

Error 3: Learning rate of Optimizer being 1 is too large.

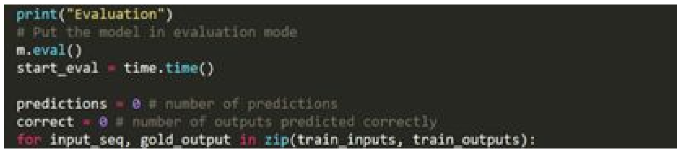

```
optimizer = optim.SGD(m.parameters(), lr=1.0)
```

Solution: change the "lr" parameter as 0.1


```
optimizer = optim.SGD(m.parameters(), lr=0.1)
```

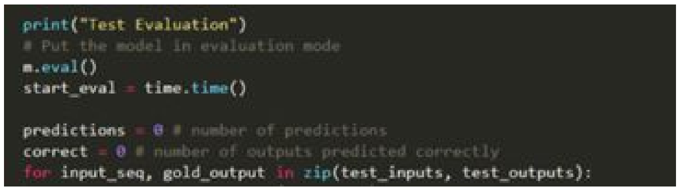
Description of Solution: Previously the training accuracy are not even optimizing with different epochs, now with learning rate of 0.1, every epoch makes improvements. If the step size is too large, the model diverges.

Error 4: The evaluation only looks at the training data not test data.


```
print("Evaluation")
# Put the model in evaluation mode
m.eval()
start_eval = time.time()

predictions = 0 # number of predictions
correct = 0 # number of outputs predicted correctly
for input_seq, gold_output in zip(train_inputs, train_outputs):
```

Solution: Add test evaluation at the end of training process. Also, test evaluation should be done after training and tuning process are done and not after each epoch.


```
print("Test Evaluation")
# Put the model in evaluation mode
m.eval()
start_eval = time.time()

predictions = 0 # number of predictions
correct = 0 # number of outputs predicted correctly
for input_seq, gold_output in zip(test_inputs, test_outputs):
```

Description of Solution:
Add test evaluation at the end of training process. Now we can know how it generalizes on the test dataset.

II. Seq2Seq

A. Toy Problems

i. Test Datasets

We constructed 4 different datasets as described below.

1. Recurrence pattern

The input parameter is recurrence degree(which indicates number of times a number is repeated. Same numbers are stored together). We experiment with four degrees of duplication which is 1,2,3,4

Evaluation Goal To analyze how model performs as a function of degree of recurrence

Dataset Creation Details:

We randomly sample numbers (length: 10/degree of recurrence) from range 0 to 19 and repeat numbers according to degree of duplication. We select the first 10 numbers to form one input of this test dataset. The procedure is repeated 1000 times.

Example:

1 1 6 6 8 8 9 9 4 4 #(Recurrence Degree=2)
7 7 7 7 3 3 3 3 2 2 #(Recurrence Degree=3)

2. Sorting order of elements

We have different test sets that are sorted in ascending order, sorted in descending order, half-sorted and odd-even arrangement of sorted numbers.

Evaluation Goal: To analyze how model performs with fully ordered, reverse ordered and partially ordered dataset

Dataset Creation Details:

We randomly sample numbers (length: 10) from range 0 to 19 without replacement. We then sort the numbers according to the pattern. (Ascending sort, Descending sort, Sorting first 5 numbers, arranging the sorted numbers at odd places and then even places). The procedure is repeated 1000 times.

Example:
1 6 7 8 10 19 2 17 9 12 #(Half-sorted)
19 17 15 14 13 10 8 7 6 5 #(Descending sorted)
1 6 2 7 3 8 4 9 5 10 #(Odd Even sorting arrangement)

3. Duplicates - Shuffled
The input parameter is Degree of duplication (which indicates number of times a number is repeated. The numbers will be shuffled later). We experiment with four degrees of duplication which is 1,2,3,4
Evaluation Goal: To analyze how model performs as a function of degree of duplication
Dataset Creation Details:
We randomly sample numbers (length: 10/degree of duplication) from range 0 to 19 and repeat numbers according to degree of duplication. We select the first 10 numbers and shuffle them to form one input of this test dataset. The procedure is repeated 1000 times.
Example:
1 6 3 2 1 3 7 7 2 6 #(Duplication Degree=2)
1 6 1 2 1 6 2 2 6 7 #(Duplication Degree=3)

4. Duplicates - Shuffled
We have different test sets for different sequence lengths i.e 5, 8, 12, 15
Evaluation Goal: To analyze how model performs as a function of sequence length of input.
Dataset Creation Details:
We randomly sample numbers (number depending on sequence length in a particular test set, 5,8,12,15) from range 0 to 19 without replacement. The procedure is repeated 1000 times.
Example:
17 16 1 7 3 #(sequence length 5)

ii. Quantitative performance on both datasets

Experiment 1
Aim: Evaluation of model performance for copy and reverse tasks as a function of degree of recurrence. (Dataset 1) (We are also reporting results for sorting task for this test sets for better analysis)
Why this dataset? As the inputs in training data do not contain any duplicates and RNNs capture the long distance context as well, we thought it would be interesting to see how model performs with duplicates that are seen consecutively

Test Set	Degree of Recurrence	Test Accuracy		
		copy task	reverse task	sort task
Given	1	0.8273	0.9337	1.0
Generated	1	0.8298	0.9272	1.0
Generated	2	0.1472	0.1083	0.4196
Generated	3	0.3454	0.2024	0.4621
Generated	4	0.3423	0.3039	0.4559

Table 1: Test Accuracy of different models on different versions on Dataset 1

Experiment 2:
Aim: Evaluation of model performance for copy and reverse tasks as a function of type of ordering. (Dataset 2) (We are also reporting results for sorting task for this test sets for better analysis)
Why this dataset? RNNs capture the long distance context and so we thought it would be interesting to see how model performs with different types of orderning. Is it actually learning the copy and reverse function irrespective of some ordering which may help us to view how model performs with unseen input.

Test Set	Type of ordering	Test Accuracy		
		copy task	reverse task	sort task
Given	random	0.8273	0.9337	1.0
Generated	asc sorted	0.8056	0.9147	1.0
Generated	desc sorted	0.8636	0.9348	1.0
Generated	first half sorted	0.8244	0.9259	1.0
Generated	sorted numbers odd-even interleaved	0.8338	0.9218	1.0

Table 2: Test Accuracy of different models on different versions on Dataset 2

Experiment 3:
Aim: Evaluation of model performance for sorting task as a function of degree of duplication. (Dataset 3) (We are also reporting results for copy and reverse task for this test sets for better analysis)
Why this dataset? As the inputs in training data do not contain any duplicates and RNNs capture the long distance context as well, we thought it would be interesting to see how model performs with duplicates arranged randomly

Test Set	Degree of Duplication	Test Accuracy		
		sort task	copy task	reverse task
Given	1	1.0	0.8273	0.9337

Generated	1	1.0	0.8325	0.9356
Generated	2	0.4227	0.3931	0.4186
Generated	3	0.4603	0.3349	0.3375
Generated	4	0.4586	0.3134	0.3459

Table 3: Test Accuracy of different models on different versions on Dataset 3

Experiment 4:
Aim: Evaluation of model performance for all toy tasks as a function of sequence length. (Dataset 4)

Test Set	**Sequence length**	**Test Accuracy**		
		sort task	**copy task**	**reverse task**
Given	10	1.0	0.8273	0.9337
Generated	5	0.6368	0.4336	0.8764
Generated	8	0.8003	0.6756	0.7162
Generated	12	0.4341	0.4496	0.4723
Generated	15	0.2458	0.2667	0.3491

Table 4: Test Accuracy of different models on different versions on Dataset 4

iii. Examples of differences in performance

Copy Model:
Given Test Set Input: 12, 14, 1, 13, 16, 15, 10, 3, 6, 8
Expected Output: 12, 14, 1, 13, 16, 15, 10, 3, 6, 8
Test Set Output: 12, 14, 1, 16, 13, 15, 10, 3, 8, 6
#Correct/Total: 6/10

Dataset 1 Input (degree = 2): 18, 18, 7, 7, 5, 5, 16, 16, 9, 9
Expected Output: 18, 18, 7, 7, 5, 5, 16, 16, 9, 9
Dataset 1 Output: 18, 7, 18, 5, 16, 7, 9, 5, 3, 16
#Correct/Total: 1/10

Reverse Model:
Given Test Set Input: 19, 12, 3, 5, 16, 17, 7, 6, 18, 10
Expected Output: 10, 18, 6, 7, 17, 16, 5, 3, 12, 19
Test Set Output: 10, 18, 6, 7, 17, 16, 5, 12, 3, 19
#Correct/Total: 8/10

Dataset 2 Input: 10, 11, 12, 13, 17, 18, 19, 2, 3, 5
Expected Output: 5, 3, 2, 19, 18, 17, 13, 12, 11, 10
Dataset 2 Output: 5, 3, 2, 19, 18, 17, 13, 10, 12, 11
#Correct/Total: 7/10

Sort Model:
Given Test Set Input: 4, 7, 6, 15, 12, 2, 3, 1, 14, 9
Expected Output: 1, 12, 14, 15, 2, 3, 4, 6, 7, 9
Test Set Output: 1, 12, 14, 15, 2, 3, 4, 6, 7, 9
#Correct/Total: 10/10

Dataset 3 Input (degree = 4): 1, 8, 8, 6, 6, 1, 6, 6, 1, 1
Expected Output: 1, 1, 1, 1, 6, 6, 6, 6, 8, 8
Dataset 3 Output: 8, 1, 1, 1, 10, 14, 6, 6, 19, 2
#Correct/Total: 5/10

iv. Justification for observed behavior

- We can see that performance of all toy tasks worsens when we used duplicates (shuffled as well as sequential, Table 1 and Table 3) which is justified as the model has not been trained using duplicates as input. Representation of hidden state might be significantly impacted when inputs have duplicates. (as the input value that contributes to that feature is doubled) Also, model might not be able to represent hidden state in better way as same number might follow two different numbers.
- Copy task is indeed not concerned about the context so much, and so using context might actually confuse the model. That might be the reason it has the lowest accuracy of all toy tasks.
- We can also see that sorting task on given testset produces accuracy 1.0 which might be because our model is actually benefited from using long distance context in case of sorting.
- We can see from Table 2 that irrespective of the order of input numbers, copy, reverse and sort task perform similarly which may be attributed to its capability to learn the respective tasks when data is not duplicated. Also, the distribution of the generated data might be similar to that of training data and the model is able to represent context without being concerned regarding any order of elements.
- Also, in general, sorting task accuracy is better than reverse task which is indeed better than copy task. This can also be attributed to the fact sorting and reverse task actually benefits from context compared to copy task. Sorting may be better as the model is able to learn the function depending on the magnitude of vector.
- We can see from table 4 that accuracy of all tasks reduces as the sequence length decrease and increases from 10 which is also justified as the model has been trained using sequence of length 10
- The models were trained on small dataset with 5 epochs which might not be sufficient time to learn the function completely and so are not performing that well on dataset 1,3,4. Increasing the size of training data to capture more cases and training for more epochs can be helpful.

B. Questions

Q1:

The task difficulties rankings are listed in following table. We can see that the ranking for the neural model we implemented is not in agreement with the other two methods.

Method	**Task Difficulties Ranking**
Human	Copy < Reverse < Sort
Deterministic Algorithm	Copy < Reverse < Sort
Neural Model	Sort < Reverse < Copy

Table 5: Task Difficulties Ranking

Reason: It takes almost no effort for human and deterministic algorithm to copy and a little more effort to reverse since both tasks can be finished in one pass, that is, O(n) computation, while sorting takes at least O(nlogn) time. But for our neural model, which learns long-term dependencies, sorting is easiest and copying is hardest because sorting encodes most context information while copying encodes almost no dependency.

Q2:

(a) The toy tasks are easier because the results of these tasks are deterministic, that is, there is a unique correct output corresponding to each input. But for sequence to sequence tasks, it is common that there are several different outputs for the same input because of language ambiguity.

(b) The sequence output for the toy tasks is some kind of permutation of the input while sequence to sequence tasks in NLP usually generate output that is completely different from the input, e.g., Machine Translation. Also, as machine learning problems, NLP tasks need a lot of training data and resource to learn a function while the toy tasks can define the function directly in the algorithm and need no training.

Q3:

The toy tasks are harder than sequence-to-sequence tasks in NLP because there is limited long-term dependency involved. In sequence-to-sequence model, the input is treated explicitly as a sequence. For example, the output for a RNN encoder is a vector representation of the input sequence and the decoder performs the same task for every element of the sequence with output being dependent on the previous computations. Since copying is independent for each element in a sequence, it is naturally more difficult for seq2seq model. The same argument holds for reversing and sorting because they rarely depend on context information.

Q4:

(a) Runtime (n is the sequence length)

Approach	Copy	Reverse	Sort
Neural Models	O(n)	O(n)	O(n)
Classical Algorithms	O(n)	O(n)	Mergesort O(nlog(n))

Table 6: Runtime Complexities for different tasks for neural models and classical deterministic algorithms

In practice, the runtime of neural model depends on different factors. For example, if batch is used, we can run neural models in parallel which will shorten the runtime significantly so it is faster than classical algorithms. Another possibility is that the structure for a neural network is over complicated and the runtime becomes much longer. Neural models also need to be trained before using them for inference. The training is of complexity O(m x n) where m is the number of training examples.

The practical run time for neural models might be much longer than deterministic algorithm. This is because the running time of neural models is related to its architecture as well. If we have a 1000 layer neural models, then it might take ages to complete inference phase. Also, the training of neural models are much longer than the inference phase. Yet, since we are using mini-batch gradient descent, we can utilize some hardware like GPUs to fast training process using parallelization.

(b) Guarantees

For classical algorithm, there is guarantee for correctness. But for neural models, there is no guarantee for correctness. The correctness depends on various factors like training data distribution and size, the consistency between testing and training dat, etc. Usually it is impossible to get 100% accuracy because the training data is not infinite and can not include all possible values in the actual distribution.

(c) Information

In the case of sequence-sorting, only the randomized embedding of inputs is provided directly to the neural model. In a standard sorting algorithm, input numbers along with the sorting function(rule) are provided to the algorithm while the neural model needs to learn the function(rule) by itself.

Q5:

The performance for RNN models with and without teacher forcing is recorded in the following table.

Task	Test Accuracy	
	With Teacher Forcing	Without Teacher Forcing
Copy	0.8275	0.5216
Reverse	0.9338	0.9457
Sort	1.0	1.0

Table 7: Test Accuracy for different models on given test set with and without teacher forcing

Explanation:
Teacher forcing uses golden label instead of the predicted output for next time step which prevents the model from off-tracking and getting punished for every subsequent word. Hence, there is a remarkable improvement for RNN model with teacher forcing on copying. As for reversing and sorting, the performance are about the same for models with and without teacher forcing because we only trained for very short time so the models would have some variation in performance.

C. Attention

i. Implementation

Attention mechanism improves on the previous experimented encoder-decoder network where all the information required are contained in the last hidden state of encoder network. It relaxes the condition and makes decoder decided on which parts of the encoding input it should focus on.

The attention mechanism is implemented by first calculating the attention weights and then the context vector.
For calculating attention weights, the steps follow as below:
The attention weight for jth time step is

$$\alpha^j = softmax(a_1^j, a_2^j, \dots, a_n^j) = softmax(f^{attn}(s_1, h_j), f^{attn}(s_2, h_j), \dots, f^{attn}(s_n, h_j))$$

s is the encoder hidden state and h is the decoder hidden states and the sequence length is n. The softmax layer is used to normalize the attention weights.
The context vector for time step j is calculated based on attention weights and encoder hidden states.

$$c_j = \sum_{i=1}^{n} a_i^j \cdot s_i$$

Then we concatenate the context vector c with the embedded input vector and fed them as input to the decoder LSTM(RNN) to precede the next time step.

$$h_{j+1} = RNN_{dec}(h_j, [embed_j; c_j])$$

For Multiplicative attention:

$$f^{attn}(s_i, h_j) = h_j^T W_a s_i \text{ ,W}_\text{a} \text{ is a trainable weight matrix}$$

For Additive attention:

$$f^{attn}(s_i, h_j) = \text{v}_\text{a} tanh(W_1 h_i + W_2 s_i)\text{, where v}_\text{a}\text{ , } W_1\text{ , } W_2 \text{ are trainable parameters}$$

The attention weights and context vector are calculated in every time step of decoding process. The LSTM function in pytorch takes input vector with shape of (1,1,64) instead of the entire sequence of shape (10,1,64). In this way, we can use a for loop to add attention mechanism since we need to modify this algorithm in each time step. We need to get access to every hidden state and to be able to concatenate a context vector to the LSTM cell at each time step as well, which entails the need to do it neatly inside a for loop.
Detailed implementation is mentioned in files main.py and model.py

ii. Experiment

Experiment 1:
To verify the correctness of attention, we did several sanity checks: (1) the attention weights add to 1 and (2) weights and context vector differ at each time step. Also, we create a special dataset with a relatively large vocabulary (0-40) where each sample is of length of 20. We compare the model with and without attention in terms of classification accuracy for the 'sorting' toy task. The result is shown below:

Test accuracy	**Model**		
	Without Attention	**Multiplicative**	**Additive**
epoch 1	**0.16**	**0.37**	**0.27**
epoch 2	**0.46**	**0.66**	**0.55**
epoch 3	**0.58**	**0.78**	**0.69**
epoch 4	**0.69**	**0.83**	**0.78**
epoch 5	**0.75**	**0.91**	**0.841**

Table8. The testing accuracy of different models for customized dataset of length 20 sequences for 'sorting' task

Test Datasets: We carried out this experiment using given test set, our synthetic data set with degree of duplication 2 and another dataset with double vocabulary and sequence length (i.e sequence length=20 and vocabulary ranges from 0 to 39).

Quantitative Performance

Experiment 2: We evaluated model performance for models trained for task copy, sort and reverse with and without attention on the given test set. We considered Additive and Multiplicative attention for our models.

Model	**Test Set Accuracy (Given Test Set)**		
	Without attention	**With Additive Attention**	**With Multiplicative Attention**
Copy	0.8273	0.9915	1.0
Reverse	0.9337	0.9373	0.9373
Sort	1.0	1.0	1.0

Table 9: Test Accuracy results for models with and without attention on given test set

Experiment 3: We evaluated model performance for models trained for task copy, sort and reverse with and without attention on dataset 3 with degree of duplication 2. We considered Additive and Multiplicative attention for our models. The vanilla models given did not perform well on dataset with duplicates. Therefore, we wanted to see if attention could improve the accuracy.

Model	**Test Set Accuracy (Dataset 3)**		
	Without attention	**With Additive Attention**	**With Multiplicative Attention**
Sort	0.4227	0.434	0.4484

Table10: Test Accuracy results for sorting model with and without attention on our dataset 3 with degree of duplication 2

Examples

Below are some examples for experiment on given test set for copy model with and without attention.

Input: 13 15 1 4 5 0 14 19 10 12
Expected Output: 13 15 1 4 5 0 14 19 10 12
Output (Without attention): 13 15 1 5 4 14 0 10 19 12
#Correct/Total: 4/10
Output (Additive Attention): 13 15 1 4 5 0 14 19 10 12
#Correct/Total: 10/10
Output (Multiplicative Attention): 13 15 1 4 5 0 14 19 10 12
#Correct/Total: 10/10

Justification

- When we extended the sequence length from 10 to 20, the text accuracy drops from 1 to 0.75. It is proved that the sequence model does not perform well on long sequence. For a relative large vocabulary and a longer input sequence to model, it is seen that the attention performs a lot better than without. It is because it can encode more 'focused' dependency on the far away input vectors to learn and represent their relationship in the learning phase. We can see similar behavior in Experiment 2 and 3 due to same reason.
- The multiplicative attention outperforms the additive attention. It has been studied that the additive is better theoretically for high dimension inputs like word embedding. One of reasons to observe a worse performance of additive attention might be that we only trained 5 epochs for the model. Since there are 3 weight matrices to optimize, the training time to the optimum solution will take longer time.

iii. Asymptotic Performance

- For the sequence models without attention, the cost is relatively cheap. The encoder is linearly dependent on the input length therefore is O(n). The decoder produces the same length of output thus the time for decoding is also O(n). Other cost like generating embedding, softmax cross entropy, could be considered as constant time operation. The overall big O without attention is O(n).
- When we add attention, encoding of input sequences remain O(n), the run time differs in the decoding phase. Now, we need a context vector in each time step. The context vector is computed by running n times of computing attention weights for each encoding states. The followed softmax normalization step and summation of n vectors could be considered as constant run time. Therefore, the complexity of decoding phase becomes $O(n^2)$ and the total running time becomes $O(n)+O(n^2)$ which is same as $O(n^2)$.
- For additive attention and multiplicative attention, they both have the same asymptotic run time which is $O(n^2)$. Yet, their practical run times differ. The multiplicative attention is faster and more space efficient since it only has simple matrix multiplication and one weight matrix to optimize. In the other hand, additive attention involves multiplication and addition and needs to store 3 parameters, making it slower and less space efficient.

III. Workflow

Task	**Person**
Debugging	Jahanvi, Tian, Xingyu
Toy Task Dataset creation	Jahanvi, Tian, Xingyu
Toy Task Experiments	Jahanvi, Tian, Xingyu
Attention Implementation	Xingyu
Attention Experiments	Jahanvi, Tian, Xingyu
Report	Jahanvi, Tian, Xingyu

IV. Feedback

Very interesting project but it would have been better if the topics were covered in class beforehand. A short demonstration of pytorch or would have been really helpful. The debugging part can also be replaced by some simple implementation of pytorch modelling.