

PROGRAMMING ASSIGNMENT 5: DEEP LEARNING

10-301/10-601 Introduction to Machine Learning (Summer 2025)

<https://www.cs.cmu.edu/~hchai2/courses/10601/>

OUT: Tuesday, June 3rd

DUE: Friday, June 6th

TAs: Andy, Canary, Michael, Sadrishya, and Neural the Narwhal

START HERE: Instructions

- **Collaboration policy:** Collaboration on solving the homework is allowed, after you have thought about the problems on your own. It is also OK to get clarification (but not solutions) from books or online resources, again after you have thought about the problems on your own. There are two requirements: first, cite your collaborators fully and completely (e.g., “Jane explained to me what is asked in Question 2.1”). Second, write your solution *independently*: close the book and all of your notes, and send collaborators out of the room, so that the solution comes from you only. See the Academic Integrity Section on the course site for more information: <https://www.cs.cmu.edu/~hchai2/courses/10601/>
- **Late Submission Policy:** See the late submission policy here: <https://www.cs.cmu.edu/~hchai2/courses/10601/>
- **Submitting your work:**
 - **Programming:** You will submit your code for programming questions on the homework to Gradescope (<https://gradescope.com>). After uploading your code, our grading scripts will autograde your assignment by running your program on a virtual machine (VM). When you are developing, check that the version number of the programming language environment (e.g. Python 3.12.9) and versions of permitted libraries (e.g. `numpy` 2.2.4, `matplotlib` 3.10.0) match those used on Gradescope. You have a **total of 10 Gradescope programming submissions**. Use them wisely. In order to not waste code submissions, we recommend debugging your implementation on your local machine (or the linux servers) and making sure your code is running correctly first before any Gradescope coding submission.
 - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. You must typeset your submission using \LaTeX . If your submission is misaligned with the template, there will be a **2% penalty** (e.g., if the homework is out of 100 points, 2 points will be deducted from your final score). Each derivation/proof should be completed in the boxes provided. Do not move or resize any of the answer boxes. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader.

For multiple choice or select all that apply questions, shade in the box or circle in the template document corresponding to the correct answer(s) for each of the questions. For \LaTeX users, replace `\choice` with `\CorrectChoice` to obtain a shaded box/circle, and don't change anything else.

Instructions for Specific Problem Types

For “Select One” questions, please fill in the appropriate bubble completely:

Select One: Who taught this course?

- ☒ Henry Chai
- ☐ Marie Curie
- ☐ Noam Chomsky

If you need to change your answer, you may cross out the previous answer and bubble in the new answer:

Select One: Who taught this course?

- ☒ Henry Chai
- ☐ Marie Curie
- ☒ Noam Chomsky

For “Select all that apply” questions, please fill in all appropriate squares completely:

Select all that apply: Which are scientists?

- ☒ Stephen Hawking
- ☒ Albert Einstein
- ☒ Isaac Newton
- ☐ I don't know

Again, if you need to change your answer, you may cross out the previous answer(s) and bubble in the new answer(s):

Select all that apply: Which are scientists?

- ☒ Stephen Hawking
- ☒ Albert Einstein
- ☒ Isaac Newton
- ☒ I don't know

For questions where you must fill in a blank, please make sure your final answer is fully included in the given space. You may cross out answers or parts of answers, but the final answer must still be within the given space.

Fill in the blank: What is the course number?

10-601

10-~~6~~301

Written Problems (23 points)

1 Empirical Questions

The following questions should be completed as you work through the programming part of this assignment. **Please ensure that all plots are computer-generated.** For all questions, unless otherwise specified, set `embed_dim`, `hidden_dim`, and `batch_size` to be 128, `num_sequences` to 50,000, and `dk`, `dv` to 128. Upload `colab_notebook.ipynb` to Google drive to make running the empirical questions easier. Please use [THIS DATA](#) for the empirical sections.

1. (4 points) First, we will experiment with changing the size / number of parameters in the model.

Generate two plots, one with the training loss and the other with the validation loss. The y-axis should have the loss value and the x-axis should have the number of sequences utilized for training so far. Each of the two plots should have four lines:

- the training/validation loss using `embed_dim = hidden_dim = 64`,
- the training/validation loss using `embed_dim = hidden_dim = 128`,
- the training/validation loss using `embed_dim = hidden_dim = 256`,
- the training/validation loss using `embed_dim = hidden_dim = 512`,

Please *include a legend* that clearly indicates which curve corresponds to which embedding and hidden dimensionalities and *please title the plot* denoting whether it is training or validation loss accordingly

[Total expected runtime on Colab T4: 15 minutes]

Your Answer

2. For this part, we will be experimenting the batch size and observing its impact on both performance during training and validation as well as speed.

(a) (4 points) Generate two plots, one with the training loss and the other with the validation loss. The y-axis should have the loss value and the x-axis should have the number of sequences utilized for training so far. Each of the two plots should have four lines:

- the training/validation loss using `batch_size = 32`,
- the training/validation loss using `batch_size = 64`,
- the training/validation loss using `batch_size = 128`,
- the training/validation loss using `batch_size = 256`,

Please *include a legend* that clearly indicates which curve corresponds to which batch size and *please title the plot* denoting whether it is training or validation loss accordingly.

[Total expected runtime on Colab T4: 20 minutes]

Your Answer

(b) (2 points) Report the total time taken for each of the batch sizes from the previous part (a) in the form of a table. The table contains four rows, one each for `batch_size` 32, 64, 128, and 256.

Batch Size	Time (sec)
32	
64	
128	
256	

- (c) (2 points) **Fill in the blank:** Complete the following statement accurately by selecting the best option for each blank, based on your observations from parts (a) and (b).

Increasing batch size leads to ____ (i) ____ training, ____ (ii) ____ in performance,
and ____ (iii) ____ stable training.

- i. ☐ faster
☐ slower
☐ no difference
 - ii. ☐ an increase
☐ a decrease
☐ no difference
 - iii. ☐ more
☐ less
☐ equally
3. (4 points) For this question, we will experiment with the number of sequences seen during training and the effects on performance.

Generate two plots, one for training loss and the other for the validation loss. The x-axis would have the `num_sequences` and y-axis would contain the *final* training/validation losses. Plot the final training/validation losses for `num_sequences=10000, 20000, 50000, and 100000`. Unlike previous questions, there will just be a single line in each plot.

[Total expected runtime on Colab T4: 15 minutes]

Your Answer

4. Finally, we will train a model on significantly more sequences and sample generations at different temperature settings to get very compelling variations of a short story.

- (a) (3 points) Fill in the table below with the final train and validation loss, and total running time of the model using the following hyperparameter setting and train the model:

- `num_sequences = 250000`,
- `batch_size = 128`,
- `embed_dim = hidden_dim = 512`,
- `key, value dimensions = 256`,

[Total expected runtime on Colab T4: 30 minutes]

Train Loss	Validation Loss	Time (sec)

- (b) (3 points) Sample three different generations from the above model and report your favorite sample for each value of `temperature` in the set $\{0, 0.3, 0.8\}$. Using the `complete` method provided in the starter code, generate completions for the prompt “Once upon a time there was a”. Include each of the completions below.

HINT: Use `torch.load()` to load the saved model after training is complete to avoid having to re-train the model for each generation.

Temperature = 0

Temperature = 0.3

Temperature = 0.8

- (c) (1 point) Describe the trend in the generated completions as `temperature` increases. Which value generated the “best” completions in your opinion?

Your Answer

2 Collaboration Questions

After you have completed all other components of this assignment, report your answers to these questions regarding the collaboration policy. Details of the policy can be found [here](#).

1. Did you receive any help whatsoever from anyone in solving this assignment? If so, include full details.
2. Did you give any help whatsoever to anyone in solving this assignment? If so, include full details.
3. Did you find or come across code that implements any part of this assignment? If so, include full details.

Your Answer

3 Programming: Language Modeling (51 points)

Large language models (LLMs) like ChatGPT, Gemini, and LLaMA have achieved unprecedented levels of success (and hype) over the past few years. In this section, you will become familiar with the building blocks of these models by implementing your very own Recurrent Neural Network LLM¹ with self-attention.

You will be building your model using PyTorch, a widely-used open source deep learning library. **In this homework, you can and should call any built-in PyTorch module (e.g., `nn.Linear`) *except* `nn.RNNCell`, `nn.RNN` and `nn.MultiheadAttention` or any "functional equivalents" of these.**

3.1 Libraries (IMPORTANT)

We will be using the following libraries *only* for this assignment. Make sure that the versions in your local environment match the ones listed here.

```
torch==2.5.1
transformers==4.44.0
numpy==2.2.4
```

You should not use `transformers` for anything other than loading in the tokenizer. In the handout, we have given a file called `requirements.txt`. In your environment, please run

```
pip install -r requirements.txt
```

3.2 Google Colab (ALSO IMPORTANT)

We recommend that you debug on your local machine, and use Colab just for answering empirical questions. The free compute credits offered by Google for running your code on a GPU are not infinite, and running out of credits will make finishing this homework much harder. **You are not expected to and definitely do not need to pay for extra compute.** If you do run out, you can wait until the next day, make a new email account, or even try running it on Kaggle. However, as long as you're not using the GPU to debug you shouldn't run into credit issues.

3.3 The Task

Language modeling is the task of assigning probabilities to texts (i.e. sequences of tokens). Language modeling is most commonly done with *autoregressive* models, which use the chain rule of probability to decompose the probability of a text $\mathbf{x} = [x_1, x_2, \dots, x_n]$ as follows:

$$P(\mathbf{x}) = P(x_1)P(x_2|x_1)P(x_3|x_1, x_2) \cdots P(x_n|x_1, \dots, x_{n-1})$$

In other words, an autoregressive language model is tasked with predicting the conditional probability of the next token given all the tokens that came before it, $P(x_{t+1}|x_1, \dots, y_x)$. In Section 5.5, we will see how these conditional probabilities will be computed by your RNN language model.

3.4 The Dataset

We will be training RNN-LMs on the [TinyStories](#) dataset, a collection of 2 million short stories with simple vocabularies generated by GPT-3.5 and GPT-4. Small language models trained on this data have been shown to have a high level of English fluency, strong storytelling abilities, and reasoning capabilities. We will be using a subset of this data. The dataset has already been processed for you in the `SentenceDataset` class, and batched in the dataloader, so you don't need to worry about loading the data. We do provide a sample of what the data looks like in the `untokenized_train_stories.json` and `untokenized_valid_stories.json`. Feel free to see what the stories look like.

¹Little language model

3.4.1 Tokenization

By their nature, ML models are unable to directly operate on text strings. As such, we have to first *tokenize* text into sequences of tokens by assigning numerical values to substrings (i.e. words, characters, punctuation, even whitespace). In this particular homework, we will be using *subword tokenization*, the kind of tokenization usually used by state-of-the-art language models like GPT-4 and LLaMA. This flavor of tokenization splits text up into subwords, which may be either full words or parts of words (for example, the “-ed” past tense suffix).

In the `my_tokenizer` directory of the handout, we have provided you with a pretrained tokenizer for the TinyStories dataset, which can be loaded using the `transformers` library:

```
>>> from transformers import AutoTokenizer
>>> tokenizer = AutoTokenizer.from_pretrained("my_tokenizer")
>>> tokenizer.encode("We like ML.")
[360, 192, 110, 1010, 1017, 103, 1]
>>> tokenizer.decode([360, 192, 110, 1010, 1017, 103, 1])
'We like ML.</s>'
```

Note that the training and validation data has already been tokenized for you. However, if you would like to convert tokens back to text, you can use the `tokenizer.decode` method, as shown above.

3.5 Required Reading: PyTorch Tutorial

Before proceeding any further, you must complete the PyTorch Tutorial. Please read the full collection of the Introduction to PyTorch, i.e. Learn the Basics || Quickstart || Tensors || Datasets & DataLoaders || Transforms || Build Model || Autograd || Optimization || Save & Load Model.

<https://pytorch.org/tutorials/beginner/basics/intro.html>

3.6 Model Definition

In this homework, you will create a language model that uses an RNN backbone to score and generate text in `rnn.py` (starter code for this file is provided in the handout).

3.6.1 RNN Cell

This is the building block of the RNN, representing a single RNN step. This class has a single method, `forward`, which takes in the batched input for the current step \mathbf{i}_t and the previous hidden state \mathbf{h}_{t-1} and outputs the next hidden state \mathbf{h}_t . Recall that the hidden state is defined as follows:

$$\mathbf{h}_t = \phi(\mathbf{W}_{i2h}\mathbf{i}_t + \mathbf{W}_{h2h}\mathbf{h}_{t-1})$$

where ϕ is an activation function, either ReLU or tanh in this assignment. **Hint:** Here (and in the rest of the model definition, including in say Section 7.5.1), matrix products like $\mathbf{W}_{i2h}\mathbf{i}_t$ correspond to `nn.Linear` layers in PyTorch.

3.6.2 Self-Attention

In `SelfAttention` you will implement scaled dot product attention. Both methods in this class will take in a sequence of vectors up to the current timestep t , $[\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_t]$. Note that the computation is batched, but for the sake of simplicity we write the equations below for a sample. In this case, the query, keys, and values

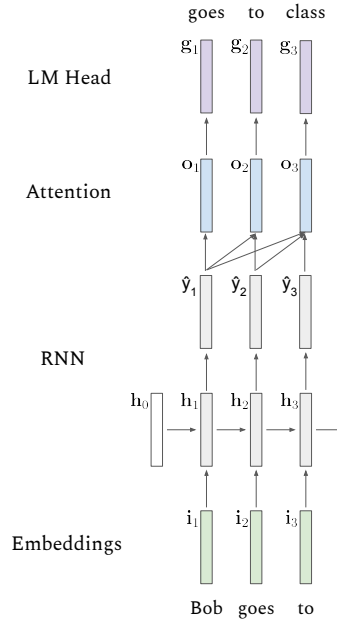


Figure 1: Computation graph of RNN with attention.

are defined as follows.

$$\begin{aligned}\mathbf{q}_t &= \mathbf{W}_q \hat{\mathbf{y}}_t \\ \mathbf{k}_t &= \mathbf{W}_k \hat{\mathbf{y}}_t \\ \mathbf{v}_t &= \mathbf{W}_v \hat{\mathbf{y}}_t\end{aligned}$$

Then, the output of the attention module will be the attention vector for the current timestep, \mathbf{a}_t , computed by a weighted average of the values. Note that we use scaled dot-product attention (as proposed in the original transformers paper), which divides the dot product of the key and query vectors by the dimension of the keys, D_{key} .

$$\mathbf{s} = \left[\frac{\mathbf{k}_1 \cdot \mathbf{q}_t}{\sqrt{D_{\text{key}}}}, \frac{\mathbf{k}_2 \cdot \mathbf{q}_t}{\sqrt{D_{\text{key}}}}, \dots, \frac{\mathbf{k}_t \cdot \mathbf{q}_t}{\sqrt{D_{\text{key}}}} \right]^T \quad (\text{Attention Scores})$$

$$\mathbf{w} = \text{Softmax}(\mathbf{s}) \quad (\text{Attention Weights})$$

$$\mathbf{a}_t = \mathbf{W}_o \left(\sum_{n=1}^t w_n \mathbf{v}_n \right)$$

For this class, you will implement two functions:

`step()`: Given the predictions for all timesteps, compute the attention for just the prediction at the current time step, $\hat{\mathbf{y}}_t$, using the above equations. Be very careful about the dimensions that you transpose in order to calculate the attention scores correctly for \mathbf{s} . At the end, we apply an additional transform to get our output \mathbf{o}_t .

`forward()`: Compute the attention for predictions across all timesteps, $[\hat{\mathbf{y}}_1, \dots, \hat{\mathbf{y}}_t]$. It is recommended to use the previously implemented `step()` function to calculate the outputs iteratively over the timesteps.

Then, concatenate all of the outputs along the sequence length (number timesteps t) dimension to get output states $[\mathbf{o}_1, \dots, \mathbf{o}_t]$

3.6.3 RNN

This class represents the entire RNN and processes an entire batched sequence $[\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_T]$. Note the distinction between this class and `RNNCell`: `RNNCell`'s `forward` method runs a single step of the input sequence, while `RNN`'s `forward` runs for all steps of the input sequence. Hence, this class contains (up to) two modules, (1) the RNN cell and (2) Linear layer to project `hidden_size` to `hidden_size`. This class has two methods, `step` and `forward`.

`step()`: This method processes a single step of the batched input sequence. Specifically, it takes both in the current input \mathbf{i}_t and all preceding hidden states $[\mathbf{h}_1, \dots, \mathbf{h}_{t-1}]$ and returns both the next hidden state \mathbf{h}_t and the next output state of the RNN $\hat{\mathbf{y}}_t$.

`forward()`: This method processes an entire batched input sequence $[\mathbf{i}_1, \mathbf{i}_2, \dots, \mathbf{i}_T]$. It should iteratively call `step` on each vector in the input sequence, along with the appropriate hidden states argument. It should return the hidden states $[\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_T]$ and output states $[\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \dots, \hat{\mathbf{y}}_T]$ over all steps.

3.6.4 RNN Language Model

Now, we can use the RNN backbone we have defined in the previous sections to create our very own LLM. However, we will need to add a couple modules on both ends of the RNN. Before the RNN, we need an embedding layer to convert integer token indices $[x_1, \dots, x_T]$ to vector embeddings $[\mathbf{i}_1, \dots, \mathbf{i}_T]$ that can be passed into the RNN as inputs. **Hint:** use `nn.Embedding` for this.

Then, we declare the RNN module, based on the size of the vector embeddings to the hidden dimension to get all of the output states to be fed at once in the Attention Layer.

Next, initialize a Self Attention layer, with inputs of hidden, key, query dimensions.

Finally, we will need to add a language modeling head, a linear layer W_{LM} that projects the RNN output \mathbf{o}_t to the next-token logits $\mathbf{g}_t = W_{\text{LM}}\mathbf{o}_t$. These logits are not actually the next-token distribution; rather, they are “raw scores” that can be fed into a softmax to yield the distribution. Hence, we have that

$$P(x_{t+1}|x_1, \dots, x_t) = \text{Softmax}(\mathbf{g}_t)$$

This module has three functions:

`forward()`: Calls all of the modules in the order as discussed above. To recap, we create embeddings from the input Tensor and then get the hidden states and predictions from the RNN forward. Call Self Attention on these predictions and then apply the LM head. Finally, return the tokens generated from the LM head and the RNN hidden/output states.

`select_token()`: This function has been implemented already. It will sample a token from the probability distribution based on the sampling policy. For greedy sampling, we just take the argmax. For temperature sampling, we need to re-compute the probability distribution and randomly sample 1 token.

`generate()`: This function has been implemented already. It will call `select_token()`, RNN, Self-Attention iteratively until we reach the max number of tokens. Note that the initial call to the model's `forward()` function computes all of the outputs and hidden states for the prefix string, then calls the `step()` function for RNN and SelfAttention because generating tokens one-by-one and don't have access to the entire sequences at once anymore.

3.7 Training and Evaluation

Now that you have created a working model, it's time to train and evaluate it! For this section, you will complete two functions: `train()` and `validate()`.

`train()`: This function will train the model, and validate the model every 10% that we process. The argument `num_sequences` specifies how long we train for, for every tenth that we progress we will validate our model. Note that this is a little different from what you are used to, since we are not using epochs as a way to control the training, instead we are controlling it by setting the number of sequences the model will process. This is so that the model can see a more diverse set of sequences as opposed to looping over the same set multiple times.

We will use the same loss function as in HW5, cross entropy loss. Here, the cross entropy is computed between the predicted next-token distribution $P(\hat{y}_{t+1}|y_1, \dots, y_t)$ and a target one-hot distribution with a 1 at the index corresponding to the true next token y_{t+1} . This loss should be computed for each token and then averaged over the sequence. Please see the `nn.CrossEntropyLoss` PyTorch documentation for instructions on how it should be used to compute the cross entropy loss.

Be careful to correctly shift the target tokens. For instance, the target for the first token is not the first token itself but the second (i.e., the next token). Also, as there is no token after the last token x_T , there can be no target for the last step's predicted next-token distribution $P(x_{T+1}|x_1, \dots, x_T)$. Thus, we can only compute loss for the first $T - 1$ tokens in the sequence.

`validate()`: This function computes the average loss over a validation dataset. This should be implemented nearly the same as the `train` method, minus the gradient updates to the model.

3.8 Text Generation

While we have been training language models to compute probabilities over texts, they are more than probability estimators! Instead of using the next-token distribution to score existing tokens, we can also use it to predict new ones.

Suppose we are given a text prefix consisting of tokens $\mathbf{x} = [x_1, \dots, x_m]$. Then when we pass the final token x_m into our language model, we will get next token logits \mathbf{g}_m which imply a distribution $P(x_{m+1}|x_1, \dots, x_m)$. In this homework, you will implement two methods of picking a next token \hat{x}_{m+1} using this distribution:

1. **Greedy:** Pick the next token with the highest probability.

$$\hat{x}_{m+1} = \operatorname{argmax}_{x_{m+1}} P(x_{m+1}|x_1, \dots, x_m)$$

2. **Temperature Sampling:** Rather than deterministically picking the highest probability next token, we can also randomly sample a next token. However, we don't necessarily have to sample from the next-token distribution $P(x_{m+1}|x_1, \dots, x_m)$ itself. Instead, we will sample from a slightly different *temperature-adjusted* distribution Q :

$$Q(x_{m+1}|x_1, \dots, x_m) = \operatorname{Softmax}(\mathbf{g}_m/\tau)$$
$$\hat{x}_{m+1} \sim Q(x_{m+1}|x_1, \dots, x_m)$$

where \mathbf{g}_m refers to the next-token logits (as defined in 5.5.4) and τ is a sampling parameter referred to as the “temperature”. The value of τ affects how “random” the samples are. Increasing the value of τ increases “randomness,” while setting $\tau = 0$ recovers greedy decoding.

After picking a token, this token can be fed back into the language model (i.e., get its embedding, feed that into the next step of the RNN, etc...) to yield another next-token distribution. This process can be repeated until some stopping criterion is met.

3.9 Command Line Arguments

The autograder runs and evaluates the output from the files generated, using the following command:

```
$ python3 rnn.py [args...]
```

Where [args...] is a placeholder for command-line arguments: <train_data> <val_data>

Additional hyper-parameters for the model utilize “double dashes”. You should experiment with these arguments to improve the performance of the model in the empirical section. <--embed_dim> <--hidden_dim>, <--dk>, <--dv>, <--num_sequences>, <--batch_size>

These arguments are described below:

1. <--train_data>: string path to the training input .txt file
2. <--val_data>: string path to the validation input .txt file
3. <--metrics_out>: string path to the ouputput .txt file to write the final train and validation loss to
4. <--train_losses_out>: string path to the output .txt file to write the training losses to
5. <--val_losses_out>: string path to the output .txt file to write the validation losses to
6. <--embed_dim> positive integer specifying the size of the sentence embedding vector (**hyper-parameter**)
7. <--hidden_dim> positive integer specifying the number of hidden units to use in the model’s hidden layer (**hyper-parameter**)
8. <--dk>: integer specifying size of the keys in self attention (**hyper-parameter**)
9. <--dv>: integer specifying size of the values in self attention (**hyper-parameter**)
10. <--num_sequences> positive integer specifying how many sequences to process (**hyper-parameter**)
11. <--batch_size> batch size of the experiment (**hyper-parameter**)

Below is an example command to run

```
python3 rnn.py --train_data data/train_stories.json \  
--val_data data/valid_stories.json \  
--train_losses_out train_losses.txt \  
--val_losses_out val_losses.txt \  
--metrics_out metrics.txt \  
--embed_dim 64 --hidden_dim 128 \  
--dk 32 --dv 32 --num_sequences 128 --batch_size 1
```

3.10 Outputs and Tests

Your code should write out a single output file (path given by the `--metrics_out` flag) containing the train and validation losses per epoch. Metrics writing is already taken care of for you in the starter code.

To help you debug your code, we've included a test file in your handout, `test_runner.py`. This is a nonexhaustive set of tests which are meant to help you make sure your implementation is correct. Passing these tests does not guarantee a full score in your Gradescope submission, but it will help you identify functions which have errors. Do not edit these tests as we will not be able to guarantee correctness if you modify these tests. To run the test, run the following command line:

```
python3 test_runner.py
```

In addition, for debugging purposes, we have included “tiny” versions of the train and validation datasets and a file `tiny_metrics.txt` which contains metrics for the following command:

```
python3 rnn.py --train_data data/tiny_train_stories.json \
--val_data data/tiny_valid_stories.json \
--train_losses_out tiny_train_losses.txt \
--val_losses_out tiny_val_losses.txt \
--metrics_out tiny_metrics.txt \
--embed_dim 64 --hidden_dim 128 \
--dk 32 --dv 32 --num_sequences 128 --batch_size 1
```

Your metrics should match these to at least 3-4 decimal places. There may be some more deviation if you run your code locally/on Colab with a GPU, but all Gradescope submissions will be run on CPU so this should not be an issue.

3.11 Gradescope Submission

You should submit your `rnn.py` (**Note:** you must submit a `.py` file, `.ipynb` files will not be processed correctly). **Any other files will be deleted.** Please do not use other file names. This will cause problems for the autograder to correctly detect and run your code.

Note: For this assignment, you have 10 submissions to Gradescope before the deadline, but only your last submission will be graded.