HOMEWORK 7: DEEP LEARNING *

10-301 / 10-601 INTRODUCTION TO MACHINE LEARNING (FALL 2026)
https://www.mlcourse.org/10601

OUT: Thursday, November 5th
DUE: Thursday, November 12th
TAs: Jayden, Julia, Sanvi, Shreyans, Neural the Narwhal

Summary In this assignment you will implement an RNN with Attention and performance evaluation. You will begin by going through some conceptual questions about CNNs, RNNs, and transformers for intuition for deep learning models and then use that intuition to build your own models.

START HERE: Instructions

- **Collaboration Policy**: Please read the collaboration policy here: https://www.mlcourse.org/10601/syllabus
- **Late Submission Policy:** See the late submission policy here: https://www.mlcourse.org/10601/syllabus
- **Submitting your work:** You will use PhiGrade to submit answers to all questions and code. Please follow instructions at the end of this PDF to correctly submit all your code to PhiGrade.
  - **Written:** For written problems such as short answer, multiple choice, derivations, proofs, or plots, please use the provided template. Submissions can be handwritten onto the template, but should be labeled and clearly legible. If your writing is not legible, you will not be awarded marks. Alternatively, submissions can be written in LaTeX. Each derivation/proof should be completed in the boxes provided. You are responsible for ensuring that your submission contains exactly the same number of pages and the same alignment as our PDF template. If you do not follow the template, your assignment may not be graded correctly by our AI assisted grader and 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).
  - **Programming:** You will submit your code for programming questions on the homework to PhiGrade. You run the autograder from your own machine: your program runs locally, and the output it produces is sent to PhiGrade, where it is compared against the reference solution. You are only permitted to use Pytorch modules (see writeup for fully authorized list), the Python Standard Library modules and `numpy`.

    Ensure that the version number of your programming language environment (i.e. Python 3.12.*) and versions of permitted libraries (i.e. `numpy` 2.2.4) match those used on PhiGrade. Every run

*Compiled on 2026-11-04 at 21:32:35

of the autograder is recorded as a submission, and your most recent one counts.

- **Materials:** The data and reference output that you will need in order to complete this assignment is posted along with the writeup and template on the course website.

Instructions for Specific Problem Types

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

Select One: Who taught this course?

- ● Matt Gormley
- ○ Henry Chai
- ○ 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?

- ● Matt Gormley
- ○ Henry Chai
- ⊗ Noam Chomsky

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

Select all that apply: Which are instructors for this course?

- ■ Matt Gormley
- ■ Bryan Wilder
- □ Henry Chai
- □ 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 the instructors for this course?

- ■ Matt Gormley
- ■ Bryan Wilder
- ⊠ Henry Chai
- ⊠ 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 Questions (59 points)

1 LaTeX Point and Template Alignment (1 points)

1.1. (1 point) **Select one:** Did you use LaTeX for the entire written portion of this homework?

- ◯ Yes
- ◯ No

1.2. (0 points) **Select one:** I have ensured that my final submission is aligned with the original template given to me in the handout file and that I haven't deleted or resized any items or made any other modifications which will result in a misaligned template. I understand that incorrectly responding yes to this question will result in a penalty equivalent to 2% of the points on this assignment.
Note: Failing to answer this question will not exempt you from the 2% misalignment penalty.

- ◯ Yes

1.3. (0 points) **Select one:** Did you fill out the Exit Poll for the previous HW? Completing the exit poll will count towards your participation grade.

- ◯ Yes

2 Convolutional Neural Network (15 points)

2.1. In this problem, consider a convolutional layer from a standard implementation of a CNN as described in lecture, **without any bias term**.

$X =$

1	0	-2	3	4	1
2	9	5	6	0	-1
0	-3	1	3	4	4
6	5	2	0	6	8
-5	4	-3	1	3	-2
4	1	2	8	9	7

$F =$

-1	-1	-1
-1	8	-1
-1	-1	-1

$Y =$

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

2.1.a. (1 point) Let an image X (6×6) be convolved with a filter F (3×3) using no padding and a stride of 1 to produce an output Y (4×4). What is value of j in the output Y?

Your Answer

2.1.b. (1 point) Suppose you instead had an input feature map (or image) of size 7×5 (height $\times$ width) and a filter of size 3×3, using a padding of 1 (i.e. a border of zeros one pixel wide is added around all four sides of the input) and a stride of 2. What would be the resulting output size? Write your answer in the format: height $\times$ width.

Your Answer

2.2. Parameter sharing is a very important concept for CNN because it drastically reduces the complexity of the learning problem and consequently that of the model required to tackle it. The following questions will deal with parameter sharing. Assume that there is no bias term in our convolutional layer.

2.2.a. (1 point) **Select all that apply:** Which of the following are parameters of a convolutional layer?

- ☐ Stride size
- ☐ Padding size
- ☐ Input size
- ☐ Filter size
- ☐ Weights in the filter

- □ Number of filters (i.e. number of output channels)
- □ None of the above

2.2.b. (1 point) **Select all that apply:** Which of the following are hyperparameters of a convolutional layer?

- □ Stride size
- □ Padding size
- □ Input size
- □ Filter size
- □ Weights in the filter
- □ Number of filters (i.e. number of output channels)
- □ None of the above

2.2.c. (1 point) Suppose for the convolutional layer, we are given grayscale images of size 22×22. Using one single 4×4 filter with a stride of 2, no padding and a single output channel, what is the **number of parameters** you are learning in this layer?

Your Answer

2.2.d. (1 point) Now suppose we do not do parameter sharing. That is, each output pixel of this layer is computed by a separate 4×4 filter. Again we use a stride of 2, no padding and a single output channel. What is the **number of parameters** you are learning in this layer?

Your Answer

2.2.e. (1 point) **Select one:** For a single 22×22 input image, which layer performs more multiplications in one forward pass: the layer from part (c) or the layer from part (d)?

- ◯ The layer from part (c)
- ◯ The layer from part (d)
- ◯ Both perform the same number of multiplications

2.2.f. (1 point) Now suppose you are given a 40×40 colored image, which consists of 3 channels, each representing the intensity of one primary color (so your input is a $40 \times 40 \times 3$ tensor). Once again, you attempt to produce an output map without parameter sharing, using a unique 4×4 filter per output pixel, with a stride of 2, no padding and a single output channel (so the number of channels in the filter are the same as the number of channels in the input image). What is the number of parameters you are learning in this layer?

Your Answer

2.2.g. (1 point) In *one concise sentence*, describe a reason why parameter sharing is a good idea for a convolutional layer applied to image data, besides the reduction in number of learned parameters.

Your Answer

2.3. Neural the Narwhal was expecting to implement a CNN for Homework 5, but he is disappointed that he only got to write a simple fully-connected neural network.

2.3.a. (2 points) Neural decides to implement a CNN himself and comes up with the following naive implementation:

```
# image X has shape (H_in, W_in), and filter F has shape (K, K)
# the output Y has shape (H_out, W_out)
Y = np.zeros((H_out, W_out))
for r in range(H_out):
    for c in range(W_out):
        for i in range(K):
            for j in range(K):
                Y[r, c] += X[___blank___] * F[i, j]
```

What should be in the *blank* above so that the output `Y` is correct? Assume that `H_out` and `W_out` are pre-computed correctly, the filter has a stride of 1 and there's no padding.

Your Answer

2.3.b. (2 points) Neural now wants to implement the backpropagation part of the network but is stuck. He decides to go to office hours to ask for help. One TA tells him that a CNN can actually be implemented using matrix multiplication. He receives the following 1D convolution example:

> Suppose you have an input vector $\mathbf{x} = [x_1, x_2, x_3, x_4, x_5]^T$ and a 1D convolution filter $\mathbf{w} = [w_1, w_2, w_3]^T$. Then if the output is $\mathbf{y} = [y_1, y_2, y_3]^T$, $y_1 = w_1x_1 + w_2x_2 + w_3x_3$, $y_2 = \cdots$, $y_3 = \cdots$. If you look at this closely, this is equivalent to
>
> $$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \mathbf{A} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$
>
> where the matrix $\mathbf{A}$ is given as $\cdots$

What is matrix $\mathbf{A}$ for this $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$? Write only the final answer. Your work will *not* be graded.

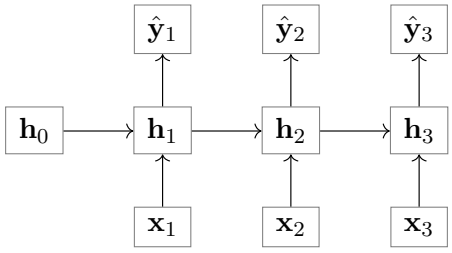
Your Answer

2.3.c. (2 points) Neural wonders why the TA told him about matrix multiplication when he wanted to write the backpropagation part. Then he notices that the gradient is extremely simple with this version of CNN. Explain in *one concise sentence (or one short mathematical expression)* how you can compute $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ once you obtain $\mathbf{A}$ for some *arbitrary* input $\mathbf{x}$, filter $\mathbf{w}$, and the corresponding 1D convolution output $\mathbf{y}$ (so $\mathbf{A}$ is obtained following the same procedure as in part (b), but $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{w}$ can be different from the example). Write only the final answer in denominator layout. Your work will *not* be graded.

Your Answer

3 Recurrent Neural Network (15 points)

3.1. Consider the following simple RNN architecture:



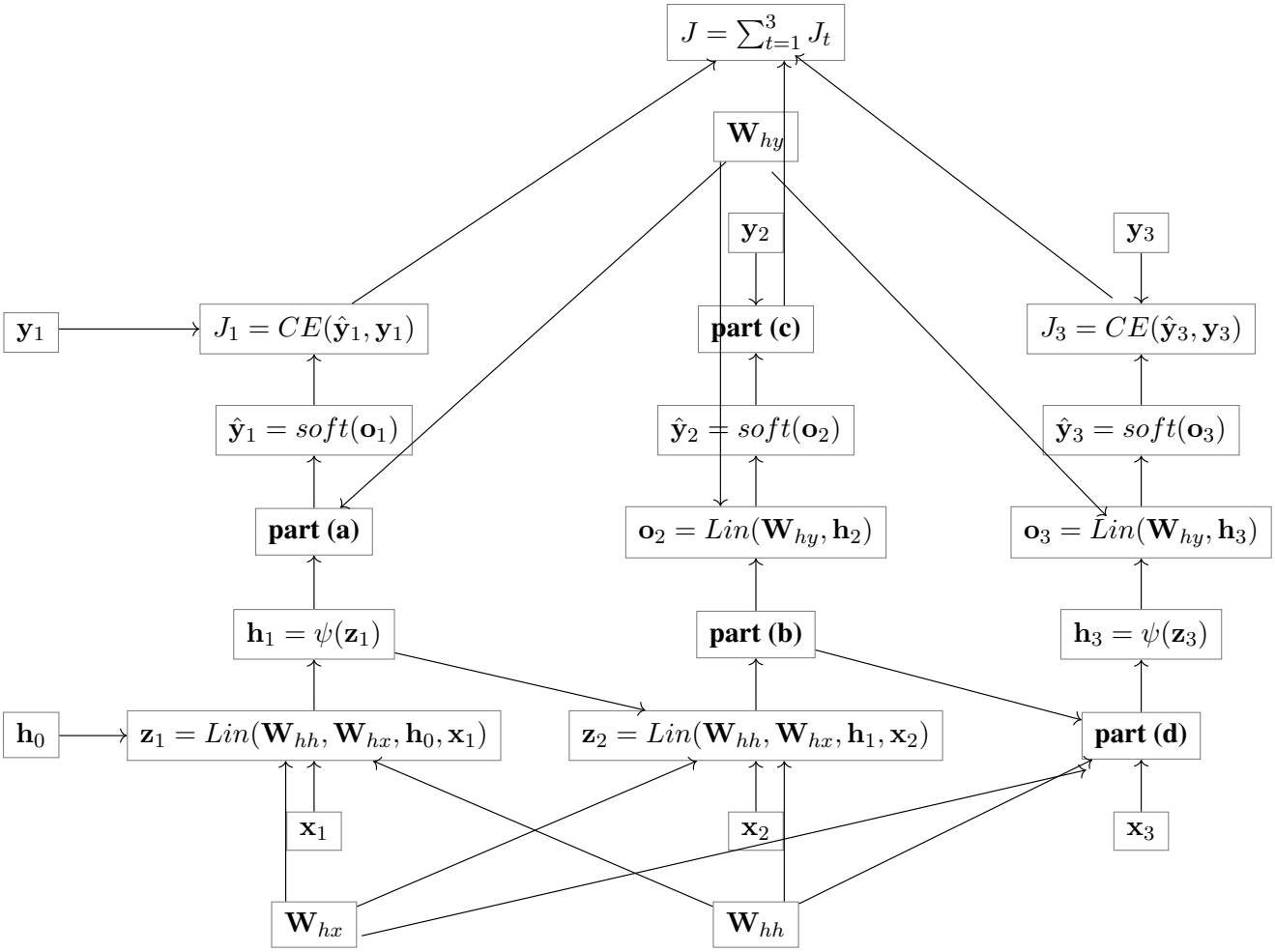
where we have inputs $\mathbf{x}_t$, hidden states $\mathbf{h}_t$, and outputs $\hat{\mathbf{y}}_t$ for each timestep t. The dimensions of these and the weights of the model are given below. On the right, we show the computation, performed by the RNN to obtain the outputs $\hat{\mathbf{y}}_t$ and subsequently the loss J for a single input $\mathbf{x}_{1:3}$.

$$\mathbf{x}_t \in \mathbb{R}^3 \qquad \mathbf{W}_{hx} \in \mathbb{R}^{4\times 3}$$
$$\mathbf{h}_t \in \mathbb{R}^4 \qquad \mathbf{W}_{hy} \in \mathbb{R}^{2\times 4}$$
$$\mathbf{y}_t, \hat{\mathbf{y}}_t \in \mathbb{R}^2 \qquad \mathbf{W}_{hh} \in \mathbb{R}^{4\times 4}$$

$$\begin{aligned}
\mathbf{z}_t &= \mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{hx}\mathbf{x}_t \\
\mathbf{h}_t &= \psi(\mathbf{z}_t) \\
\mathbf{o}_t &= \mathbf{W}_{hy}\mathbf{h}_t \\
\hat{\mathbf{y}}_t &= soft(\mathbf{o}_t) \\
J_t &= -\sum_{i=1}^{2} y_{t,i} \log(\hat{y}_{t,i}) \\
J &= \sum_{t=1}^{3} J_t
\end{aligned}$$

Above $\mathbf{y}_t$ is a one-hot vector representing the label for the tth timestep, $soft$ is the **softmax** activation, ψ is the **identity** activation (i.e. no activation), J is the cross entropy loss computed by the function $CE()$. Note here that we assume that we have no intercept term.

3.1.a. (4 points) You will now construct the unrolled computational graph for the given model. Use input sequence $\mathbf{x}$, label $\mathbf{y}$, and the RNN equations presented above to complete the graph by filling in the solution boxes for the corresponding blanks.



(a)

(b)

(c)

(d)

3.1.b. Now you will derive the steps of the backpropagation algorithm that lead to the computation of $\frac{dJ}{d\mathbf{W}_{hh}}$. For all parts of this question, please write your answer in terms of $\mathbf{W}_{hh}$, $\mathbf{W}_{hy}$, $\mathbf{y}$, $\hat{\mathbf{y}}$, $\mathbf{h}$, and any additional terms specified in the question (note: this does not mean that every term listed shows up in every answer, but rather that you should simplify terms into these as much as possible when you can).

3.1.b.i. (2 points) What is $g_{J_t} = \frac{\partial J}{\partial J_t}$? Write your solution in the first box, and show your work in the second.

$\frac{\partial J}{\partial J_t}$

Work

3.1.b.ii. (2 points) What is $g_{\mathbf{o}_t} = \frac{\partial J}{\partial \mathbf{o}_t}$ for an arbitrary $t \in [1, 3]$? Write your solution in the first box, and show your work in the second. Write your answer in terms of $\hat{\mathbf{y}}_t$, $\mathbf{y}_t$, and g_{J_t}. (Hint: Think about how you can write J_t in terms of $\mathbf{o}_t$, then use the chain rule. You may want to use a result from homework 5 to help here.)

$\frac{\partial J}{\partial \mathbf{o}_t}$

Work

3.1.b.iii. (3 points) What is $g_{\mathbf{h}_i} = \frac{\partial J}{\partial \mathbf{h}_i}$ for an arbitrary $i \in [1, 3]$? Write your solution in terms of $\mathbf{g}_{\mathbf{o}_t}$, $\mathbf{W}_{hh}$, $\mathbf{W}_{hy}$ in the first box, and show your work in the second. (Hint: Find $\frac{\partial \mathbf{o}_t}{\partial \mathbf{h}_i}$, then use the chain rule. Also, for a given i, think about which $\mathbf{o}_t$'s $\mathbf{h}_i$ affects)

$\frac{\partial J}{\partial \mathbf{h}_i}$

Work

3.1.b.iv. (2 points) What is $g_{\mathbf{W}_{hh}} = \frac{\partial J}{\partial \mathbf{W}_{hh}}$? Write your solution in terms of $\mathbf{g}_{\mathbf{h}_i}$ and $\mathbf{h}_i$ in the first box, and show your work in the second.
Hint: $\mathbf{g}_{\mathbf{h}_i}$ already captures the unrolled backward/future gradients for timestep i, so you only need to consider the contribution of $\mathbf{W}_{hh}$ at each timestep locally

$\frac{\partial J}{\partial \mathbf{W}_{hh}}$

Work

3.2. (2 points) **Select all that apply:** Which of the following are true about Recurrent Neural Net and Recurrent Neural Net Language Model?

- ☐ An RNN cannot process sequential data, whereas an RNN-LM is designed for sequential data processing such as in natural language processing.
- ☐ An RNN-LM is only exclusively used as an encoder, which can process sequential data and encode it into a fixed-size state vector.
- ☐ An RNN-LM includes additional layers and structures specifically designed to predict the next token in a sequence, making it more suited for tasks like text generation than a standard RNN.
- ☐ The RNN-LM is trained to maximize the probability of a sequence of tokens, given a previous sequence, which is not a typical training objective of a standard RNN.
- ☐ None of the above.

4 Transformers and AutoDiff (5 points)

```
global tape = stack()

class Module:

    method init()
        out_tensor = null
        out_gradient = 1

    method apply_fwd(List in_modules)
        in_tensors = [x.out_tensor for x in in_modules]
        out_tensor = forward(in_tensors)
        tape.push(self)
        return self

    method apply_bwd():
        in_gradients = backward(in_tensors, out_tensor, out_gradient)
        for i in 1,..., len(in_modules):
            in_modules[i].out_gradient += in_gradients[i]
        return self

function tape_bwd():
    while len(tape) > 0
        m = tape.pop()
        m.apply_bwd()
```

4.1. (1 point) **Select one:** This is a code snippet from lecture 19 slide 68. In the context of the `method apply_fwd()` inside the `Module` class, what is the primary role of the `tape.push(self)` command that pushes the module onto the tape?

◯ It records the current module onto the stack along with its parameters and tensors to ensure that the output tensor is saved for the backward pass.

◯ It pushes the current computation's gradient onto the stack for immediate use in the forward pass.

◯ It duplicates the module to allow for parallel computations in subsequent layers of the neural network.

◯ It activates the module for the forward pass, making it the only active computation in the network.

4.2. (2 points) **True or False:** We can replace a stack with a queue in Module-based AutoDiff without increasing the runtime of the algorithm. Explain your reasoning in no more than 2 sentences in the box below.

◯ True

◯ False

Your Answer

4.3. Consider a Transformer model employing a multi-headed self-attention mechanism. Suppose the input consists of a sequence of T tokens, each token represented by a d_{model}-dimensional embedding vector. This model utilizes H attention heads. During the attention process, each head generates keys, queries, and values from the input embeddings. The dimensionality of the key and query vectors is d_k for each head, and the attention function produces an output vector of dimension d_v for each token and head.

4.3.a. (1 point) Which of the following represents the dimension of the key tensor for a single attention head?

◯ $T \times d_v$

◯ $H \times d_k \times d_{\text{model}}$

◯ $T \times d_k$

◯ $T \times d_{\text{model}} \times d_k$

4.3.b. (1 point) Which of the following represents the dimension of the output tensor of the multi-headed attention before any final linear transformation?

◯ $T \times (H \times d_k)$

◯ $T \times (H \times d_v)$

◯ $T \times d_{\text{model}}$

◯ $(H \times d_k) \times d_v$

5 Empirical Questions (23 points)

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. Please upload `colab_notebook.ipynb` to Google drive to make running the empirical questions easier

5.1. (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; A100: 10 minutes]

Your Answer

5.2. 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.

5.2.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, A100; 15 minutes]

Your Answer

5.2.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. Report time taken with either a T4 or an A100.

Batch Size	**Time (sec)**
32	
64	
128	
256	

5.2.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 and _____(ii)_____ in performance.

5.2.c.i.
- ◯ faster
- ◯ slower
- ◯ no difference

5.2.c.ii.
- ◯ an increase
- ◯ a decrease
- ◯ no difference

5.3. (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; A100 : 10 minutes]

Your Answer

5.4. 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.

5.4.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; A100: 8 minutes]

Train Loss	Validation Loss	Time (sec)

5.4.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 `generate` command provided in the notebook, generate completions for the prompt "Once upon a time there was a". Include each of the completions below.

```
Temperature = 0
```

```
Temperature = 0.3
```

```
Temperature = 0.8
```

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

Your Answer

6 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

7 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 LLM[1] 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.

7.1 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 `environment.yml`. Please run

```
conda env create -f environment.yml
```

This will create the `601-hw7` environment, which also includes `phigrade` and `pytest`. You should be using this environment when working on this homework locally.

7.2 Google Colab (ALSO IMPORTANT)

We recommend using Google Colab to test your code and answer the empirical questions throughout this assignment. You may also test your code locally. The PhiGrade tests always run on your own machine, so run them from your local `601-hw7` environment. 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 homework 7 much harder. **You are not expected to and definitely do not need to pay for extra compute.** Google Colab is also offering a free year of Colab Pro to students. This will give you access to more compute hours and better GPUs. You can find more info here. If you do end up running 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. **TLDR: Test on CPU on Google Colab and then answer questions with a GPU on Google Colab.**

7.3 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 $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ as follows:

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

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, $P(x_{t+1}|x_1, \ldots, y_x)$. In Section 5.5, we will see how these conditional probabilities will be computed by your RNN language model.

[1] *Little* language model

7.4 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.

7.4.1 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.

7.5 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

7.6 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).

7.6.1 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:

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

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

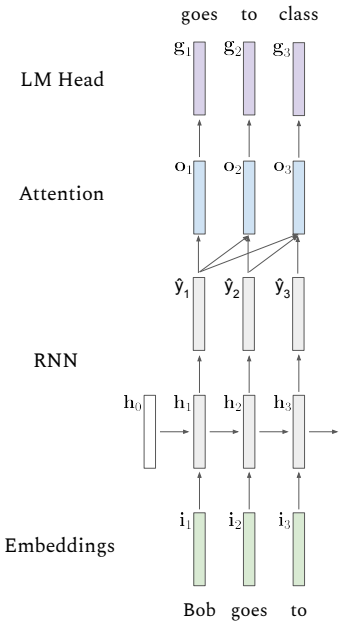


Figure 1: Computation graph of hybrid RNN with attention.

7.6.2 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, $[\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_t]$. 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 are defined as follows.

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

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 square root of the dimension of the keys, D_{key}.

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

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

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

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, $[\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_t]$. 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 $[\mathbf{o}_1, \ldots, \mathbf{o}_t]$

7.6.3 RNN

This class represents the entire RNN and processes an entire batched sequence $[\mathbf{i}_1, \mathbf{i}_2, \ldots \mathbf{i}_T]$. 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 $[\mathbf{h}_1, \ldots \mathbf{h}_{t-1}]$ 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 $[\mathbf{i}_1, \mathbf{i}_2, \ldots \mathbf{i}_T]$. 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 $[\mathbf{h}_1, \mathbf{h}_2, \ldots, \mathbf{h}_T]$ and output states $[\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, \ldots, \hat{\mathbf{y}}_T]$ over all steps.

7.6.4 Hybrid 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 $[x_1, \ldots, x_T]$ to vector embeddings $[\mathbf{i}_1, \ldots, \mathbf{i}_T]$ 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

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

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.

7.7 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 a helper function, `compute_loss()` and two main 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 $P(\hat{y}_{t+1}|y_1, \ldots, y_t)$ 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 $P(x_{T+1}|x_1, \ldots, x_T)$. 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.

7.8 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 $\mathbf{x} = [x_1, \ldots, x_m]$. 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 $P(x_{m+1}|x_1, \ldots, x_m)$. In this homework, you will be given two methods of picking a next token $\hat{x}_{m+1}$ using this distribution:

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

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

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 $P(x_{m+1}|x_1, \ldots, x_m)$ itself. Instead, we will sample from a slightly different *temperature-adjusted* distribution Q:

$$\begin{aligned} Q(x_{m+1}|x_1, \ldots, x_m) &= \text{Softmax}(\mathbf{g}_m/\tau) \\ \hat{x}_{m+1} &\sim Q(x_{m+1}|x_1, \ldots, x_m) \end{aligned}$$

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.

7.9 Command Line Arguments

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

```
$ python3 rnn.py [train/generate] [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>`

Each command (train or generate) has a different set of arguments to be passed in. The arguments for train are described below:

1. `<--train_data>`: string path to the training input `.json` file
2. `<--val_data>`: string path to the validation input `.json` file
3. `<--metrics_out>`: string path to the ouptput `.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)**
12. `<--model_path>` string path to the `.pt` file where model will be saved after training.

The arguments for generate are described below:

1. `<--model_path>` string path to the `model.pt` file where model will be loaded from.
2. `<--prefix>` Prefix string to be generated from
3. `<--num_tokens>` Number of tokens to generate (**hyper-parameter**)
4. `<--temperature>` Temperature of decoding (**hyper-parameter**)

Below is an example command to run

```
python3 rnn.py train --train_data data/small_train_stories.json \
--val_data data/small_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 \
--model_path my_model.pt
```

7.10 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 on PhiGrade, 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 --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 \
--model_path small_model.pt
```

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 the PhiGrade test suite always runs your code on CPU, even if your machine has a GPU, so this should not be an issue.

7.11 Submission Instructions

Please ensure you have completed the following files for submission.

`rnn.py`

Submission happens through PhiGrade by running the test suite from inside the handout folder. There is no file upload. Create an API token on your PhiGrade account page, then, from the handout folder:

```
$ conda env create -f environment.yml   # once
$ conda activate 601-hw7
$ export PHIGRADE_API_KEY="<YourAPIToken>"
```

```
$ pytest test_rnn_language_model.py
```

Your code runs locally and only its output is uploaded, so `rnn.py` must sit in the folder you run `pytest` from, alongside `data/`. You must submit a `.py` file; `.ipynb` files will not be processed correctly. Please do not use other file names.

The graded suite includes five end-to-end training runs on CPU that together take about 1.5 minutes.

Every run of `pytest` is a submission, and your most recent one counts. The first run asks which slot to submit to: **Slot A** (for human work only) or **Slot B** (for human or AI work). Set `PHIGRADE_SLOT` to skip the prompt on later runs.