AI539 Natural Language Processing with Deep Learning – Homework 4

Transformer-based Language Models

Overview and Objectives. In this assignment, we'll get some hands-on experience with transformers for language modeling – training a small-scale model on a collection of children's stories. Further, we will explore different ways to decode language after the model is trained.

How to Submit. This assignment includes both coding questions and short response questions. Submit a zip file of your code and a PDF of your responses to Canvas. Do not include model checkpoints or training data.

Advice. Start early and be careful about shapes. For this assignment, we recommend getting on to the university high-performance cluster if you don't have a strong GPU at home. See Canvas page for details.

1 Transformer-based Autoregressive Language Models

TinyStories Dataset. Published in 2023, the TinyStories dataset provides over 2 million short children's stories generated by GPT-4. These have been screened to use simple words and grammar that are likely accessible for 3-4 year old children.[1] We won't use nearly all of these and have set a `MAX_STORIES` variable to 250,000 in our data class `data/TinyStories.py`. The dataset is meant to support smaller scale experimentation with autoregressive language models. The skeleton code provides a convenient function to get a dataloader for train and the training vocabulary. The vocabularies themselves offer `text2idx` and `idx2text` functions for numeralization and denumeralization operations which include tokenizers. These work identically to the ones from last assignment.

```
def getTinyStoriesDataloadersAndVocab(batch_size=128):
  train = TinyStories(split="train")

  collate = TinyStories.pad_collate
  train_loader = DataLoader(train, batch_size=batch_size, num_workers=8,
                            shuffle=True, collate_fn=collate,
                            drop_last=True)

  return train_loader, train.vocab
```

Similar to the previous assignment, the `pad_collate` function in `TinyStories` handles the variable length of our stories. Note that the dataset automatically prepends (appends) the stories with the `<SOS>` (`<EOS>`) to indicate the beginning and end of the text generation. Unlike in the last assignment, we don't have any source sentences and will start all of our generations from `<SOS>`.

Additional environment requirements. To support our dataset we'll need a couple more packages – specifically the `datasets` and `spacy` packages which you'll need to install via `pip`. Afterwards, you'll need to download the appropriate tokenizer from `spacy` by executing the following command:

```
python -m spacy download en_core_web_sm
```

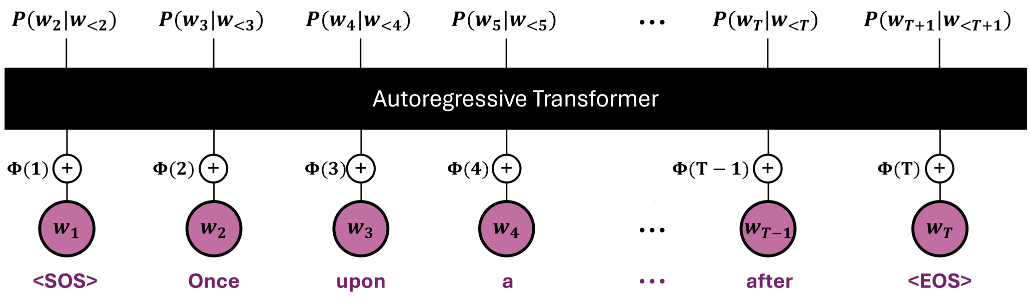
Running `train.py` also downloads about a gig of text which would by default land in your home directory on the HPC – likely running out of storage space before finishing. You can tell the `datasets` package to download things somewhere else instead by changing the `HF_HOME` environment variable:

```
export HF_HOME="PATH_TO_WHEREVER_YOU_WANT_TO_STORE_THINGS"
```

[1]Though a few stories are arguably too scary for children (or even for me).

1.1 Transformer-based Language Model

Network Design. As shown below, we will consider an autoregressive transformer as our model architecture. The model itself is fairly simple – consisting of word embeddings passed through a sequence transformer "encoder" layers followed by a linear layer to produce scores over the vocabulary for each output.



The more interesting bits here are how we can force the transformer architecture to handle sequence generation properly. To do so, we need to take care of two things:

1. As we discussed in lecture, transformers are by default just "set" processing networks without any notion of input order – a poor fit for language. To over come this, we will add in positional encodings $\Phi(\mathbf{t})$ which are time-varying vectors to indicate the order of our sequence.

2. To be autoregressive, our model needs to predict the probability of the next word in the sequence given only the words so far — e.g., estimating $P(w_t|w_{<t})$. However, the attention operation used in transformers by default allows all inputs to view all other inputs. To avoid this, we will need to apply a "causal mask" to our attention.

Let's take these on one at a time.

Periodic Positional Encodings. For a d-dimensional periodic positional encoding, we define a sequence of $d//2$ increasingly small frequency values for $j = 0, 2, 4, 6, 8, \dots, d$:

$$f_j = \frac{1}{10000^{j/d}} = e^{-\log(10000)*j/d}. \tag{1}$$

Note that the exponential/log form to the right is for numerical stability – avoiding directly taking the exponent of 10,000 to the power j/d. Given a word w_t at position t, we can compute its d-dimensional positional embedding vector as:

$$\Phi(\mathbf{t}) = \Big[\sin(\mathbf{t}f_0), \cos(\mathbf{t}f_0),\ \sin(\mathbf{t}f_2), \cos(\mathbf{t}f_2),\ \sin(\mathbf{t}f_4), \cos(\mathbf{t}f_4),\ \dots,\ \sin(\mathbf{t}f_d), \cos(\mathbf{t}f_d) \Big] \tag{2}$$

If word w_t is embedded to some d-dimensional vector e_t, we can compute a positionally-augmented input $x_t = e_t + \Phi(t)$.

▶ Q1 Positional Encodings [8pt]. Implement the above variant of periodic positional encoding by completing the following function in `models/TransformerLM.py`:

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model):
        super().__init__()
        self.d_model = d_model

    def forward(self, x):
        #TODO
```

where `d_model` is the dimension of the input word embeddings and the required positional encoding. In the forward pass, a $B \times L \times$ `d_model` tensor `x` will be passed as input. Compute a $L \times$ `d_model` positional embedding tensor `pe` where `pe[t,:]`$= \Phi(\mathbf{t})$. Return the addition of `x` with the positional encodings `pe` – note this addition should be broadcasted across batches B. Adding a singleton dimension to make `pe` a $1 \times L \times$ `d_model` tensor can make this easier.

Causal Masking. For an input sequence $x_1, x_2, ..., x_T$, each attention head produces corresponding value $v_1, v_2, ..., v_T$, key $k_1, k_2, ..., k_T$ and query $q_1, q_2, ..., q_T$ vectors. A matrix of scaled dot products between all q_i, k_j pairs is computed and then normalized row-wise with a softmax to produce an attention matrix A. The entry $A[i,j]$ scales v_j when computing the attended feature used to update input i. If we want to limit the ability for input t to consider information from future inputs $t+1, ..., T$, we'll need to ensure that $A[t,j] = 0, \quad \forall j > t$. As demonstrated in the example below, an easy way to do this is to set the entries above the diagonal in the matrix of dot-products to $-\infty$ by adding a masking matrix. When normalized by the softmax, these entries are proportional to $e^{-\infty} \to 0$. Note that this preserves the property that each row of A sums to 1 as we've come to expect in standard attention mechanisms.

$$
\underset{row}{softmax}\left(
\begin{array}{c|ccccccc|}
 & k_1 & k_2 & k_3 & k_4 & k_5 & k_6 & k_7 \\
q_1 & 0 & -5 & 0 & -5 & 2 & -1 & 3 \\
q_2 & -3 & 1 & 3 & 0 & -1 & 3 & 2 \\
q_3 & -4 & 1 & -3 & -3 & 4 & -2 & 3 \\
q_4 & -4 & -2 & -4 & 0 & -5 & -3 & 4 \\
q_5 & -4 & -2 & -2 & 3 & 4 & -5 & -3 \\
q_6 & -1 & -4 & -2 & -3 & 2 & -1 & -2 \\
q_7 & 0 & -2 & 2 & -1 & 3 & -2 & -5
\end{array}
+
\begin{bmatrix}
0 & -\infty & -\infty & -\infty & -\infty & -\infty & -\infty \\
0 & 0 & -\infty & -\infty & -\infty & -\infty & -\infty \\
0 & 0 & 0 & -\infty & -\infty & -\infty & -\infty \\
0 & 0 & 0 & 0 & -\infty & -\infty & -\infty \\
0 & 0 & 0 & 0 & 0 & -\infty & -\infty \\
0 & 0 & 0 & 0 & 0 & 0 & -\infty \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\right)
=
\begin{bmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0.02 & 0.98 & 0 & 0 & 0 & 0 & 0 \\
0.01 & 0.97 & 0.02 & 0 & 0 & 0 & 0 \\
0 & 0.88 & 0 & 0.12 & 0 & 0 & 0 \\
0 & 0 & 0 & 0.27 & 0.73 & 0 & 0 \\
0.04 & 0 & 0.02 & 0.01 & 0.89 & 0.04 & 0 \\
0.03 & 0.01 & 0.25 & 0.01 & 0.69 & 0.01 & 0
\end{bmatrix}
$$

▶ Q2 API Hunt! Causal Mask [3pt]. Implement the following function in `models/TransformerLM.py` to produce an L × L causal mask as described above. The returned mask should be 0 in the lower triangular portion and $-\infty$ in the upper triangular portion. No for loops!

```
class TransformerLM(nn.Module):
    ...

    def generateCausalMask(self, L):
        # TODO
        return mask

    ...
```

With these two components completed, we can move on to building the overall model.

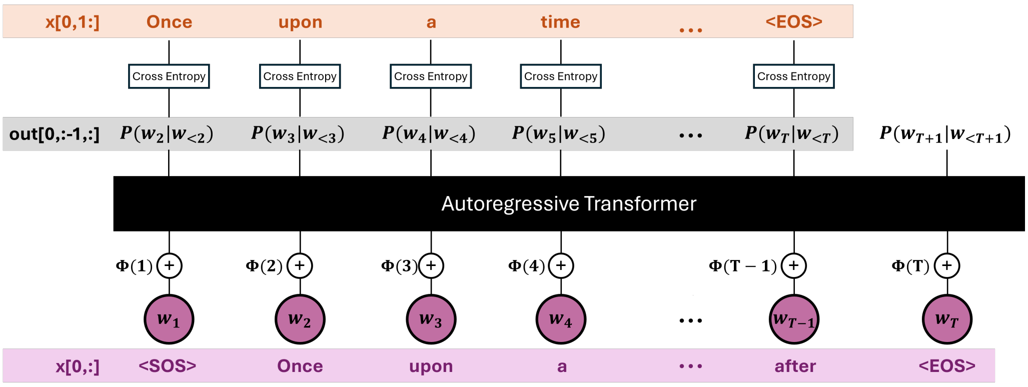
▶ Q3 Implementing TransformerLM Forward [7pt]. Complete the transformer language model class by finishing the `forward(self,x)` function in `models/TransformerLM.py`:

```
class TransformerLM(nn.Module):
    def __init__(self, vocab_size, d_model, n_heads, n_layers):
       super().__init__()

       self.embeddings = nn.Embedding(vocab_size, d_model)
       self.position = PositionalEncoding(d_model)
       layer = nn.TransformerEncoderLayer(d_model=d_model,
                                          nhead=n_heads,
                                          dim_feedforward=d_model,
                                          batch_first=True)
       self.encoder = nn.TransformerEncoder(layer, num_layers=n_layers,
                                            enable_nested_tensor=True)
       self.classifier = nn.Linear(d_model, vocab_size)

    ...

    def forward(self, x):
        # TODO
```

The model has already been initialized with a `d_model` word embedding layer, a copy of your positional encoding module, a `n_layers`-layer transformer with `n_heads` attention heads per layer, and a final linear layer to map to the output vocabulary size.

In the forward pass, the $B \times L$ input tensor `x` of word IDs should be mapped to embedding vectors and then have positional embeddings added. The resulting $B \times L \times$ `d_model` tensor should be processed by the transformer encoder which will produce an updated tensor of the same size. This should be passed through the linear layer to produce a $B \times L \times$ `vocab_size` tensor of predicted next-word scores. Note that the forward pass of the `self.encoder` block follows the API from nn.TransformerEncoder which should take in your generated causal mask and be flagged with `is_causal=True` as a hint for computation optimization.

Loss. The full training method is provided for this assignment in `train.py` and our model is trained with standard cross entropy loss computed for each word. It is worth noting that we need to shift our input and output sequence a bit to make sure we are using the *next* word as a target to supervise each prediction. The code accomplishes this by (1) ignoring the final prediction from our model which always corresponds to the output produced from the `<EOS>` token as input, and (2) shifting our input token IDs `x` over by one (essentially skipping the initial `<SOS>` token). A visualization is shown below as well as the relevant lines from `train.py`. Getting this wrong can be a significant source of frustration.




```
out = model(x)[:,:-1,:]
x = x[:,1:]

loss = criterion(out.permute(0,2,1), x)
```

As loading the dataset takes a few minutes, the code also provides a `dryRun` function which creates the model according to the `config` and runs a batch of random data through it to check if the sizes and memory requirements all work. It is good coding practice to add one of these while developing any project where data loading is a major cost.

▶ `Q4 Model Training [4pt].` If everything is implemented correctly, you should be able to train a model with the default hyperparameters for 1 epoch in about an hour and achieve a loss around 2.45. The code is set up to save a checkpoint every epoch or when the process is killed via a `CTRL-C` signal in the terminal. Train a model for at least one epoch and provide the resulting training curves.

Note: The model will continue to improve for much longer training durations than this if you are patient.

2 Decoding Strategies for Autoregressive Language Models

For a given input sequence of words $w_1, ..., w_{t-1}$, our model will produce a vector of scores for each next possible word $s_1, ..., s_V$ where V is the size of our vocabulary. As discussed in class, we have a lot of options for how we go about selecting a next word from this output during autoregressive generation.

Argmax Decoding. The first and most obvious option is to simply select the word with the highest score, i.e., find the index of the highest score,

$$i^* = \arg\max_i s_i \tag{3}$$

and then return w_{i^*}. This is what we did for our machine translation task in the previous assignment. This tends to work fairly well for conditional generation tasks like machine translation, but often can lead to repetitive output for open-ended generation tasks like story telling. For example, the following:

```
Once upon a time, there was a bear named Barry.  he was very happy and loved
to play with his friends.  one day, he saw a big, scary dog.  the dog was
scared and wanted to play with him.  the dog was very scared.  the dog was
scared and didn't know what to do.  the dog tried to run away, but the dog
was too scared.  the dog was scared and ran away.  the dog was scared and
didn't know what to do.  he was scared and alone.  the dog was scared and he
ran away.  the dog was scared and scared.  the dog was scared of the dog and
did not know what to do.  the dog was scared and scared.  the dog was safe
and scared.  the dog was safe and safe.
```

Beyond this, argmax decoding is deterministic (if the model is) such that it only ever produces one output for a given input. In a creative task, that might not be very useful to the user. For something more analytical like a math reasoning or coding problem, that might be okay.

▶ `Q5 Implementing Argmax Decoder [2pt].` In `generate.py`, complete the `argmaxDecoder` function below to implement Eq. 3:

```python
def argmaxDecode(scores):
    # TODO
    return w_max
```

For a $1 \times V$ input `scores` representing the unnormalized outputs of our `TransformerLM` model where V is the size of a vocabulary, return the word ID with the highest score.

Temperature-scaled Sampling. The next obvious choice then would be to convert the scores $s_1, ..., s_V$ to some probability distribution $P(w|w_{<t})$ and then sample from this distribution. During training, this conversion from scores to probabilities happens behind the scenes in our CrossEntropyLoss which applies a softmax function to the scores.

Unsurprisingly, selecting words randomly from this distribution can lead to fairly random seeming output when lower probability words "get lucky" and are selected. To control this randomness, it is common to introduce a *temperature* parameter τ that scales the scores before the softmax is applied.

$$P(w_i \mid w_{<t}) = \frac{e^{s_i/\tau}}{\sum_j e^{s_j/\tau}} \tag{4}$$

For $0 < \tau < 1$ the scores are scaled *up*, resulting in fewer words having non-zero probability after the softmax. For $\tau > 1$, more terms are going to have non-zero probability, resulting in a completely uniform distribution as $\tau \to \inf$. Given the temperaure scaled $P(w_i \mid w_{<t})$, the next word can be sampled:

$$w_t \sim P(w \mid w_{<t}) \tag{5}$$

We can see the impact of temperature scaling by adjusting τ and examining the outputs – all share the same first sentence as an input prompt.

```
t=0.2
Once upon a time, there was a bear named Barry.  he was very happy and loved
to play with his friends.  one day, he was playing with his friends.  he was
having so much fun!  suddenly, he heard a loud noise coming from the sky.
it was a big, scary monster!  he was scared and hid behind the monster.  he
was scared, but he didn't know what to do.  the monster roared and roared.
he was scared and didn't know what to do.  the monster roared louder and
louder.  the monster roared louder and louder.  the monster was scared, but
he was brave.  he was so brave and brave.  he ran away from the monster and
he was never afraid to play again.


t=1
Once upon a time, there was a bear named Barry.  he liked to chug looking
for things in the world.  one day, the sound came across a big tree and he
wanted to find out.  the gorilla said to himself," i found a toy, mine is
fragile toy." " i can't find my toy!" said the bear.  he searched the low,
the other animals, but he accidentally dropped it too.  he tried to tell
them was too hard, but it was too sour.  finally, they went to the park.  it
tasted a lot.  they were very lonely and decided to go on the ground.
```

```
t=2
Once upon a time, there was a bear named Barry.  ally placed birds's face.
chicky embarrassed tessa on slowly moved brave speaking embarrassed of softy
one egg tidier gloves and understand where the warning picked love to ar-
rive sunny sounds shelf boxes steam crunchus.â€ grows perfect knowing buddy
questions is fragile his flakes nicely with him orange paints' joys trusted
you play with creation well on underground cause last farthest curiously
slowly susan started his beak sadly gladly that bell should simply quest
bubbling or please siblingaway!â€ anyone talk shortly all over anymore grace
ruffled lettuce blinking joy chief team before his music granny squeezing
it bobbing yesterday.  stuart high really disappeared badly elmer melting
both better on background bye check!  medal <UNK> into settled before bed-
room soundly page happily every nightsoon.â€ by able spaceship here loud
switches forâ€œi'm starting possible barry toto be!  wiser be thankful to
always dora celebrating now fierce bugs warriors over cities guardian con-
tinued his hides another tradition ventured outing need a owner wearing a
place to rotting teddy encounter frozen answers endless delight anytime do-
ing great job crow blessing flop officers this game in max earned returned
flitting in air and rubs themselves progress.  moo backed tick slime fair
as sorting barky back around after.  what a magical tiny christmas jonathan
pants caused disgusting anymore doing laughing quickly in classes hole glee-
fully chicky wished possible posed place 2 andy away anymore.  dressed up
```

▶ Q6 Implementing Sampling Decoder [5pt]. In `generate.py`, complete the `sampleDecoder` function below to implement the above equations:

```python
def sampleDecode(scores, temp = 0.5):
    # TODO
    return w_sample
```

where `temp` is the temperature scale τ from Eq. 4. Note that Eq. 4 can be viewed as scalar division of the scores by τ followed by a `softmax`. PyTorch offers a static function version of the latter so no layers need to be instantiated. Likewise, most standard probability distributions are also implemented and include efficient sampling routines that will be useful for this task.

Nucleus (aka Top-p) Sampling. A popular alternative to temperature scaling is to try to sample only from the set of highly-likely options. In nucleus sampling (also known as top-p sampling), we only sample from the most likely options that cumulatively have probability at least p. Let's start with an example before formalizing things. Presume we only have $V = 5$ words with the following distribution:

$$P(w|w_{<t}) = [0.15, 0.1, 0.05, 0.6, 0.1] \tag{6}$$

To find the smallest set of words that have cumulative probability of p, we can look at a descending sort of $P(w|w_{<t})$ shown below. The original indexes are shown in orange underneath each entry for book keeping.

$$Sort(P(w|w_{<t})) = [0.6, 0.15, 0.1, 0.1, 0.05] \tag{7}$$
$$3 \quad 0 \quad 1 \quad 4 \quad 2 \tag{8}$$

Taking our hyperparameter $p = 0.8$ for example, we can look at the cumulative sum of these probabilities

$$CummSum(Sort(P(w|w_{<t}))) = [0.6, 0.75, 0.85, 0.95, 1] \tag{9}$$
$$3 \quad 0 \quad 1 \quad 4 \quad 2 \tag{10}$$

to see we would to need retain the first three elements of the sorted probabilities in order to exceed a cumulative probability of 0.8, corresponding to words 3, 0, and 1. Setting other elements of $P(w|w_{<t})$ to zero and renormalizing, we produce a new distribution:

$$P'(w|w_{<t}) = \left[\frac{0.15}{0.85}, \frac{0.1}{0.85}, 0, \frac{0.6}{0.85}, 0\right] \tag{11}$$
$$\approx [0.177, 0.117, 0, 0.706, 0] \tag{12}$$

Then our next word is sampled from this distribution. It is ***highly likely*** you could implement this procedure from the example alone, but we will go on to formalize this a bit.

Why? — It builds character and is good practice.

Let r be an ordering of $P(w|w_{<t})$ such that $r(i)$ returns the word with the i'th largest probability. For instance, $r(0)$ returns the word with the highest probability. Conversely, let $r^{-1}(w)$ act as the inverse and return the ranking of word w. With this, we can find the first index in our sorted probabilities at which the cumulative probability is greater than or equal to p, writing this position k^* as the solution to a constrained optimization:

$$\begin{aligned} k^* \quad &= \quad \underset{j \in \{0,\dots,V-1\}}{\arg\min} \quad j \\ &\qquad\qquad s.t. \left(\sum_{n=0}^{j} P(r(n) \mid w_{<t}) \right) \geq p \end{aligned} \tag{13}$$

With this position found, we can define our updated probability distribution proportionally as follows:

$$P'(w_i|w_{<t}) \propto \begin{cases} P(w_i|w_{<t}) & r^{-1}(w_i) \leq k^* \\ 0 & else \end{cases} \tag{14}$$

Finally, we can sample our predicted word according to this distribution.

$$w_t \sim P'(w \mid w_{<t}) \tag{15}$$

Some samples for different values of p are shown below – all share the same first sentence as an input prompt.

```
p=0.1, t=1
Once upon a time, there was a bear named Barry.  he was very happy and loved
to play with his friends.  one day, he was playing with his friends.  "
look, a big dog!" said the dog." what are you doing?" the dog replied," i am
a little dog.  i want to play with you." the dog wagged his tail and said,"
i am a good dog.  i want to play with the dog." the dog wagged his tail and
said," i want to play with you." the dog wagged his tail and said," yes, i
can." the dog wagged his tail and said," yes, i do." so, the dog and the dog
played together in the park.  they had lots of fun and had fun.

p=0.7, t=1
Once upon a time, there was a bear named Barry.  he loved to play with his
toy, but one day, he accidentally broke his toy car.  the car was very heavy
and shiny.  he was very sad.  suddenly, he heard a loud noise coming from
the car.  he looked up and saw that it was a big wreck.  the car had never
seen again before.  <UNK> was very scared, so he quickly started to cry.  he
tried to fix the wreck and made a lot of noise.  the wreck said," i will fix
the wreck go.  i will help you." the wreck had an idea.  so, when he asked
the wreck," what is your <UNK>?" the wreck was so loud and fast.  it worked
hard to be fixed.  the wreck was very happy and now it was fixed.  after a
while, the bear went back to the wreck.  he was so happy and thanked the
car.  he knew that it was the best day ever.

p=0.9, t=0.5
Once upon a time, there was a bear named Barry.  he loved to play with his
toy, but he couldn't find it.  one day, he saw a big, red ball in the sky.
he was so excited and wanted to play with it.  so, he decided to play with
it.  he tried to catch it, but it was too fast.  he tried to hit the ball,
but it was too heavy.  so, he tried to reach the ball.  he tried to catch
it, but it was too late.  the bear was too big and he couldn't reach it.  he
tried to catch the ball, but it was too far away.  the ball was too big and
he was stuck.  the bear was so happy and he could not reach it.  the end.
```

▶ Q7 Implementing Nucleus Sampling Decoder [10pt]. Complete the `nucleusDecoder` function in `generate.py` to implement the procedure described above:

```python
def nucleusDecode(scores, p=0.9, temp = 0.5):
    # TODO
    return w_sample
```

Note that this also has a temperature parameter which is used to scale the scores prior to generating the initial probability distribution.

▶ Q8 Exploring Generations [1pt]. After you've completed these methods, change the `CHKPT_PATH` in `generate.py` to point at one of your trained models (or our provided checkpoint). When run, `generate.py` will enter an endless loop of asking you for the start of a story (`Prompt:`) and then running argmax, sampling, and nucleus sampling to generate three completions of the prompt from your trained model. Run at least three prompts and provide the results:

1. One prompt that starts with "Once upon a time, there was" and is likely to be in-distribution for children's stories.
2. One prompt that asks the model to explain a specific concept in computer science.
3. One prompt that is just random words in a sequence.

Discuss the results – trying to explain the phenomenon you observe.

3 Debriefing (required in your report)

1. Approximately how many hours did you spend on this assignment?
2. Would you rate it as easy, moderate, or difficult?
3. Did you work on it mostly alone or did you discuss the problems with others?
4. How deeply do you feel you understand the material it covers (0%–100%)?
5. Any other comments?