XCS224N Problem Set 5 Self-attention, Transformers, Pretraining

Due Sunday, August 29 at 11:59pm PT.

Guidelines

1. If you have a question about this homework, we encourage you to post your question on our Slack channel, at http://xcs224n-scpd.slack.com/

2. Familiarize yourself with the collaboration and honor code policy before starting work.

3. For the coding problems, you must use the packages specified in the provided environment description. Since the autograder uses this environment, we will not be able to grade any submissions which import unexpected libraries.

Submission Instructions

Written Submission: Some extra credit questions in this assignment require a written response. For these questions, you should submit a PDF with your solutions online in the online student portal. As long as the PDF is legible and organized, the course staff has no preference between a handwritten and a typeset LaTeX submission. If you wish to typeset your submission and are new to LaTeX, you can get started with the following:

- Type responses only in `submission.tex`.
- Submit the compiled PDF, **not** `submission.tex`.
- Use the commented instructions within the `Makefile` and `README.md` to get started.

Coding Submission: Some questions in this assignment require a coding response. For these questions, you should submit **all files indicated in the question** to the online student portal. For further details, see Writing Code and Running the Autograder below.

Honor code

We strongly encourage students to form study groups. Students may discuss and work on homework problems in groups. However, each student must write down the solutions independently, and without referring to written notes from the joint session. In other words, each student must understand the solution well enough in order to reconstruct it by him/herself. In addition, each student should write on the problem set the set of people with whom s/he collaborated. Further, because we occasionally reuse problem set questions from previous years, we expect students not to copy, refer to, or look at the solutions in preparing their answers. It is an honor code violation to intentionally refer to a previous year's solutions. More information regarding the Stanford honor code can be foudn at https://communitystandards.stanford.edu/policies-and-guidance/honor-code.

Note. Here are some things to keep in mind as you plan your time for this assignment.

- There are math questions again!

- The total amount of pytorch code to write, and code complexity, of this assignment is lower than Assignment 4. However, you're also given less guidance or scaffolding in how to write the code.

- This assignment involves a pretraining step that takes approximately 3 hours to perform on Azure, and you'll have to do it twice.

This assignment is an investigation into Transformer self-attention building blocks, and the effects of pretraining. It covers mathematical properties of Transformers and self-attention through written questions. Further, you'll get experience with practical system-building through repurposing an existing codebase. The assignment is split into a coding part and an extra credit written (mathematical) part. Here's a quick summary:

1. **Extending a research codebase:** In this portion of the assignment, you'll get some experience and intuition for a cutting-edge research topic in NLP: teaching NLP models facts about the world through pretraining, and accessing that knowledge through finetuning. You'll train a Transformer model to attempt to answer simple questions of the form "Where was person [x] born?" – without providing any input text from which to draw the answer. You'll find that models are able to learn some facts about where people were born through pretraining, and access that information during fine-tuning to answer the questions.

   Then, you'll take a harder look at the system you built, and reason about the implications and concerns about relying on such implicit pretrained knowledge.

2. **Mathematical exploration:** What kinds of operations can self-attention easily implement? Why should we use fancier things like multi-headed self-attention? This section will use some mathematical investigations to illuminate a few of the motivations of self-attention and Transformer networks.

1 Pretrained Transformer models and knowledge access

You'll train a Transformer to perform a task that involves accessing knowledge about the world – knowledge which isn't provided via the task's training data (at least if you want to generalize outside the training set). You'll find that it more or less fails entirely at the task. You'll then learn how to pretrain that Transformer on Wikipedia text that contains world knowledge, and find that finetuning that Transformer on the same knowledge-intensive task enables the model to access some of the knowledge learned at pretraining time. You'll find that this enables models to perform considerably above chance on a held out development set.

The code you're provided with is a fork of Andrej Karpathy's minGPT. It's nicer than most research code in that it's relatively simple and transparent. The "GPT" in minGPT refers to the Transformer language model of OpenAI, originally described in this paper [1].

As in previous assignments, you will want to develop on your machine locally, then run training on Azure. You can use the same conda environment from previous assignments for local development, and the same process for training on Azure (see the *Practical Guide for Using the VM* section of the XCS224N Azure Guide for a refresher). Specifically, you'll still be running "conda activate XCS224N" on the Azure machine. You'll need around 6 or more hours for training, so budget your time accordingly!

Your work with this codebase is as follows:

(a) **[0 points (Coding)] Check out the demo.**
In the src/submission/mingpt-demo/ folder is a Jupyter notebook (play_char.ipynb) that trains and samples from a Transformer language model. Take a look at it (locally on your computer and you might need to install Jupyter notebootk pip install jupyter) to get somewhat familiar with how it defines and trains models. Some of the code you're writing below will be inspired by what you see in this notebook.

Note that you do not have to write any code or submit written answers for this part.

(b) **[0 points (Coding)] Read through** NameDataset **in** src/submission/dataset.py, **our dataset for reading name-birth place pairs.**
The task we'll be working on with our pretrained models is attempting to access the birth place of a notable person, as written in their Wikipedia page. We'll think of this as a particularly simple form of question answering:

> Q: Where was [person] born?
> A: [place]

From now on, you'll be working with the src/submission folder. **The code in** mingpt-demo/ **won't be changed or evaluated for this assignment.** In dataset.py, you'll find the the class NameDataset, which reads a TSV (tab-separated values) file of name/place pairs and produces examples of the above form that we can feed to our Transformer model.

To get a sense of the examples we'll be working with, if you run the following code, it'll load your NameDataset on the training set birth_places_train.tsv and print out a few examples.

```
cd src/submission
python dataset.py namedata
```

Note that you do not have to write any code or submit written answers for this part.

(c) **[0 points (Coding)] Implement finetuning (without pretraining).**
Take a look at run.py. It has some skeleton code specifying flags you'll eventually need to handle as command line arguments. In particular, you might want to *pretrain*, *finetune*, or *evaluate* a model with this code. For now, we'll fxfocus on the finetuning function, in the case without pretraining.

Taking inspiration from the training code in the play_char.ipynb file, write code to finetune a Transformer model on the name/birth place dataset, via examples from the NameDataset class. For now, implement the case without pretraining (i.e. create a model from scratch and train it on the birth-place prediction task from part (b)). You'll have to modify two sections, marked [part c] in the code: one to initialize the model, and one to finetune it. Note that you only need to initialize the model in the case labeled "vanilla" for now (later

in section (g), we will explore a model variant). Use the hyperparameters for the `Trainer` specified in the `run.py` code.

Also take a look at the *evaluation* code which has been implemented for you. It samples predictions from the trained model and calls `evaluate_places()` to get the total percentage of correct place predictions. You will run this code in part (d) to evaluate your trained models.

This is an intermediate step for later portions, including Part (d), which contains commands you can run to check your implementation. No written answer is required for this part.

(d) **[10 points (Coding)] Make predictions (without pretraining).**
Train your model on `wiki_places_train.tsv`, and evaluate on `birth_dev.tsv`. Specifically, you should now be able to run the following three commands:

```
# Train on the names dataset
cd src/submission
python run.py finetune vanilla wiki.txt \
        --writing_params_path vanilla.model.params \
        --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set, writing out predictions
python run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.model.params \
        --eval_corpus_path birth_dev.tsv \
        --outputs_path vanilla.nopretrain.dev.predictions

# Evaluate on the test set, writing out predictions
python run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.model.params \
        --eval_corpus_path birth_test_inputs.tsv \
        --outputs_path vanilla.nopretrain.test.predictions
```

Training will take less than 10 minutes (on Azure). Report your model's accuracy on the dev set (as printed by the second command above). Don't be surprised if it is well below 10%; we will be digging into why in Part 3. As a reference point, we want to also calculate the accuracy the model would have achieved if it had just predicted "London" as the birth place for everyone in the dev set. Fill in `london_baseline.py` to calculate the accuracy of that approach and report your result in your write-up. You should be able to leverage existing code such that the file is only a few lines long.

(e) **[16 points (Coding)] Define a *span corruption* function for pretraining.**
In the file `src/dataset.py`, implement the `__getitem__()` function for the dataset class `CharCorruptionDataset`. Follow the instructions provided in the comments in `dataset.py`. Span corruption is explored in the T5 paper [2]. It randomly selects spans of text in a document and replaces them with unique tokens (noising). Models take this noised text, and are required to output a pattern of each unique sentinel followed by the tokens that were replaced by that sentinel in the input. In this question, you'll implement a simplification that only masks out a single sequence of characters.

This question will be graded via autograder based on your whether span corruption function implements some basic properties of our spec. We'll instantiate the `CharCorruptionDataset` with our own data, and draw examples from it.

To help you debug, if you run the following code, it'll sample a few examples from your `CharCorruptionDataset` on the pretraining dataset `wiki.txt` and print them out for you.

```
cd src/submission
python dataset.py charcorruption
```

No written answer is required for this part.

(f) **[16 points (Coding)] Pretrain, finetune, and make predictions. Budget 2 hours for training.**
Now fill in the *pretrain* portion of `run.py`, which will pretrain a model on the span corruption task. Addition-

ally, modify your *finetune* portion to handle finetuning in the case *with* pretraining. In particular, if a path to a pretrained model is provided in the bash command, load this model before finetuning it on the birth-place prediction task. Pretrain your model on `wiki.txt` (which should take approximately two hours), finetune it on `NameDataset` and evaluate it. Specifically, you should be able to run the following three four commands: (Don't be concerned if the loss appears to plateau in the middle of pretraining; it will eventually go back down.)

```
# Pretrain the model
cd src/submission
python run.py pretrain vanilla wiki.txt \
        --writing_params_path vanilla.pretrain.params

# Finetune the model
python run.py finetune vanilla wiki.txt \
        --reading_params_path vanilla.pretrain.params \
        --writing_params_path vanilla.finetune.params \
        --finetune_corpus_path birth_places_train.tsv

# Evaluate on the dev set; write to disk
python run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.finetune.params \
        --eval_corpus_path birth_dev.tsv \
        --outputs_path vanilla.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python run.py evaluate vanilla wiki.txt  \
        --reading_params_path vanilla.finetune.params \
        --eval_corpus_path birth_test_inputs.tsv \
        --outputs_path vanilla.pretrain.test.predictions
```

Report the accuracy on the dev set (printed by the third command above). We expect the dev accuracy will be at least 10%, and will expect a similar accuracy on the held out test set.

(g) **[11 points (Coding)] Research! Write and try out the *synthesizer* variant (Budget 2 hours for pretraining!)**

We'll now go to changing the Transformer architecture itself – specifically, the self-attention module. While we've been using a self-attention scoring function based on dot products, this involves a rather intensive computation that's quadratic in the sequence length. This is because the dot product between ℓ^2 pairs of word vectors is computed in each computation. *Synthesized attention* [3] is a very recent alternative that has potential benefits by removing this dot product (and quadratic computation) entirely.

It's a promising idea, and one way for us to ask, "What's important/right about the Transformer architecture, and where can we improve/prune aspects of it?" In `attention.py`, implement the `forward()` method of `SynthesizerAttention`, which implements a variant of the Synthesizer proposed in the cited paper.

The provided `CausalSelfAttention` implements the following attention for each head of the multi-headed attention: Let $X \in \mathbb{R}^{\ell \times d}$ (where ℓ is the block size and d is the total dimensionality, d/h is the dimensionality per head.).[1] Let $Q, K, V \in \mathbb{R}^{d \times d/h}$. Then the output of the self-attention head is

$$Y_i = \text{softmax}\left(\frac{(XQ_i)(XK_i)^\top}{\sqrt{d/h}}\right)(XV_i) \tag{1}$$

where $Y_i \in \mathbb{R}^{\ell \times d/h}$. Then the output of the self-attention is a linear transformation of the concatenation of the heads:

$$Y = [Y_1; \ldots; Y_h]A \tag{2}$$

were $A \in \mathbb{R}^{d \times d}$ and $[Y_1; \ldots; Y_h] \in \mathbb{R}^{\ell \times d}$. The code also includes dropout layers which we haven't written here. We suggest looking at the provided code and noting how this equation is implemented in PyTorch.

Your job is to implement the following variant of attention. Instead of Equation 1, implement the following in `SynthesizerAttention`:

$$Y_i = \text{softmax}\big(\text{ReLU}(XA_i + b_1)B_i + b_2\big)(XV_i), \tag{3}$$

where $A_i \in \mathbb{R}^{d \times d/h}$, $B_i \in \mathbb{R}^{d/h \times \ell}$, and $V_i \in \mathbb{R}^{d \times d/h}$.[2] One way to interpret this is as follows: The term $(XQ_i)(XK_i)^\top$ is an $\ell \times \ell$ matrix of attention scores, computed as all pairs of dot products between word embeddings. The synthesizer variant eschews the all-pairs dot product and directly computes the $\ell \times \ell$ matrix of attention scores by mapping each d-dimensional vector of each head for X to an ℓ-dimesional vector of unnormalized attention weights.

In the rest of the code in the `src/` folder, modify your model to support using either `CausalSelfAttention` or `SynthesizerAttention`. Add the ability to switch between these attention variants depending on whether "vanilla" (for causal self-attention) or "synthesizer" (for the synthesizer variant) is selected in the command line arguments (see the section marked `[part g]` in `src/run.py`). You are free to implement this functionality in any way you choose, so long as it supports these command line arguments.

Below are bash commands that your code should support in order to pretrain the model, finetune it, and make predictions on the dev and test sets. Note that the pretraining process will take approximately 2 hours.

```
# Pretrain the model
cd src/submission
python run.py pretrain synthesizer wiki.txt \
        --writing_params_path synthesizer.pretrain.params


# Finetune the model
python run.py finetune synthesizer wiki.txt \
        --reading_params_path synthesizer.pretrain.params \
        --writing_params_path synthesizer.finetune.params \
        --finetune_corpus_path birth_places_train.tsv


# Evaluate on the dev set; write to disk
```

---

[1] Note that these dimensionalities do not include the minibatch dimension.

[2] Hint: copy over the `CausalSelfAttention` class, and modify it minimally for this.

```
python run.py evaluate synthesizer wiki.txt  \
        --reading_params_path synthesizer.finetune.params \
        --eval_corpus_path birth_dev.tsv \
        --outputs_path synthesizer.pretrain.dev.predictions

# Evaluate on the test set; write to disk
python run.py evaluate synthesizer wiki.txt  \
        --reading_params_path synthesizer.finetune.params \
        --eval_corpus_path birth_test_inputs.tsv \
        --outputs_path synthesizer.pretrain.test.predictions
```

Report the accuracy of your synthesizer attention model on birth-place prediction on `birth_dev.tsv` after pretraining and fine-tuning.

(a) We'll score your model as to whether it gets at least 5% accuracy on the test set, which has answers held out.

2   Considerations in pretrained knowledge

(a) **[0.50 points (Written, Extra Credit)]** Succinctly explain why the pretrained (vanilla) model was able to achieve an accuracy of above 10%, whereas the non-pretrained model was not.

(b) **[0.50 points (Written, Extra Credit)]** Take a look at some of the correct predictions of the pretrain+finetuned vanilla model, as well as some of the errors. We think you'll find that it's impossible to tell, just looking at the output, whether the model *retrieved* the correct birth place, or *made up* an incorrect birth place. Consider the implications of this for user-facing systems that involve pretrained NLP components. Come up with two reasons why this indeterminacy of model behavior may cause concern for such applications.

(c) **[1 point (Written, Extra Credit)]** If your model didn't see a person's name at pretraining time, and that person was not seen at fine-tuning time either, it is not possible for it to have "learned" where they lived. Yet, your model will produce *something* as a predicted birth place for that person's name if asked. Concisely describe a strategy your model might take for predicting a birth place for that person's name, and one reason why this should cause concern for the use of such applications.

3   Attention exploration

Multi-headed self-attention is the core modeling component of Transformers. In this question, we'll get some practice working with the self-attention equations, and motivate why multi-headed self-attention can be preferable to single-headed self-attention.

Copying in attention:

(a) **[1 point (Written, Extra Credit)] Copying in attention:** Recall that attention can be viewed as an operation on a query $q \in \mathbb{R}^d$, a set of value vectors $\{v_1, \ldots, v_n\}, v_i \in \mathbb{R}^d$, and a set of key vectors $\{k_1, \ldots, k_n\}, k_i \in \mathbb{R}^d$, specified as follows:

$$c = \sum_{i=1}^{n} v_i \alpha_i \tag{4}$$

$$\alpha_i = \frac{\exp(k_i^\top q)}{\sum_{j=1}^{n} \exp(k_j^\top q)}. \tag{5}$$

where α_i are frequently called the "attention weights", and the output $c \in \mathbb{R}^d$ is a correspondingly weighted average over the value vectors.

We'll first show that it's particularly simple for attention to "copy" a value vector to the output c. Describe (in one sentence) what properties of the inputs to the attention operation would result in the output c being approximately equal to v_j for some $j \in \{1, \ldots, n\}$. Specifically, what must be true about the query q, the values $\{v_1, \ldots, v_n\}$ and/or the keys $\{k_1, \ldots, k_n\}$?

(b) **[1 point (Written, Extra Credit)] An average of two:** Consider a set of key vectors $\{k_1, \ldots, k_n\}$ where all key vectors are perpendicular, that is $k_i \perp k_j$ for all $i \neq j$. Let $\|k_i\| = 1$ for all i. Let $\{v_1, \ldots, v_n\}$ be a set of arbitrary value vectors. Let $v_a, v_b \in \{v_1, \ldots, v_n\}$ be two of the value vectors. Give an expression for a query vector q such that the output c is approximately equal to the average of v_a and v_b, that is, $\frac{1}{2}(v_a + v_b)$.[3] Note that you can reference the corresponding key vector of v_a and v_b as k_a and k_b.

(c) **[2 points (Written, Extra Credit)] Drawbacks of single-headed attention:** In the previous part, we saw how it was *possible* for a single-headed attention to focus equally on two values. The same concept could easily be extended to any subset of values. In this question we'll see why it's not a *practical* solution. Consider a set of key vectors $\{k_1, \ldots, k_n\}$ that are now randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, where the means μ_i are known to you, but the covariances Σ_i are unknown. Further, assume that the means μ_i are all perpendicular; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

   (a) (1 point) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small α. Design a query q in terms of the μ_i such that as before, $c \approx \frac{1}{2}(v_a + v_b)$, and provide a brief argument as to why it works.

   (b) (1 point) Though single-headed attention is resistant to small perturbations in the keys, some types of larger perturbations may pose a bigger issue. Specifically, in some cases, one key vector k_a may be larger or smaller in norm than the others, while still pointing in the same direction as μ_a. As an example, let us consider a covariance for item a as $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small α (as shown in figure 1). Further, let $\Sigma_i = \alpha I$ for all $i \neq a$.

   When you sample $\{k_1, \ldots, k_n\}$ multiple times, and use the q vector that you defined in part i., what qualitatively do you expect the vector c will look like for different samples?

(d) **[2 points (Written, Extra Credit)] Benefits of multi-headed attention:** Now we'll see some of the power of multi-headed attention. We'll consider a simple version of multi-headed attention which is identical to single-headed self-attention as we've presented it in this homework, except two query vectors (q_1 and q_2) are defined, which leads to a pair of vectors (c_1 and c_2), each the output of single-headed attention given its respective query vector. The final output of the multi-headed attention is their average, $\frac{1}{2}(c_1 + c_2)$. As in question 1((c)), consider a set of key vectors $\{k_1, \ldots, k_n\}$ that are randomly sampled, $k_i \sim \mathcal{N}(\mu_i, \Sigma_i)$, where

---

[3]Hint: while the softmax function will never *exactly* average the two vectors, you can get close by using a large scalar multiple in the expression.
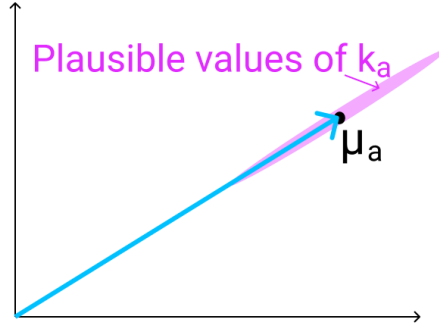
Figure 1: The vector μ_a (shown here in 2D as an example), with the range of possible values of k_a shown in red. As mentioned previously, k_a points in roughly the same direction as μ_a, but may have larger or smaller magnitude.

the means μ_i are known to you, but the covariances Σ_i are unknown. Also as before, assume that the means μ_i are mutually orthogonal; $\mu_i^\top \mu_j = 0$ if $i \neq j$, and unit norm, $\|\mu_i\| = 1$.

(a) (1 point) Assume that the covariance matrices are $\Sigma_i = \alpha I$, for vanishingly small α. Design q_1 and q_2 such that c is approximately equal to $\frac{1}{2}(v_a + v_b)$.

(b) (1 points) Assume that the covariance matrices are $\Sigma_a = \alpha I + \frac{1}{2}(\mu_a \mu_a^\top)$ for vanishingly small α, and $\Sigma_i = \alpha I$ for all $i \neq a$. Take the query vectors q_1 and q_2 that you designed in part i. What, qualitatively, do you expect the output c to look like across different samples of the key vectors? Please briefly explain why. You can ignore cases in which $q_i^\top k_a < 0$.

(e) **[2 points (Written, Extra Credit)] Key-Query-Value self-attention in neural networks:** So far, we've discussed attention as a function on a set of key vectors, a set of value vectors, and a query vector. In Transformers, we perform *self-attention*, which roughly means that we draw the keys, values, and queries from the same data. More precisely, let $\{x_1, \ldots, x_n\}$ be a sequence of vectors in $\mathbb{R}^d$. Think of each x_i as representing word i in a sentence. One form of self-attention defines keys, queries, and values as follows. Let $V, K, Q \in \mathbb{R}^{d \times d}$ be parameter matrices. Then

$$v_i = V x_i \quad i \in \{1, \ldots, n\} \tag{6}$$
$$k_i = K x_i \quad i \in \{1, \ldots, n\} \tag{7}$$
$$q_i = Q x_i \quad i \in \{1, \ldots, n\} \tag{8}$$

Then we get a context vector for each input i; we have $c_i = \sum_{j=1}^n \alpha_{ij} v_j$, where α_{ij} is defined as $\alpha_{ij} = \frac{\exp(k_j^\top q_i)}{\sum_{\ell=1}^n \exp(k_\ell^\top q_i)}$. Note that this is single-headed self-attention.

In this question, we'll show how key-value-query attention like this allows the network to use different aspects of the input vectors x_i in how it defines keys, queries, and values. Intuitively, this allows networks to choose different aspects of x_i to be the "content" (value vector) versus what it uses to determine "where to look" for content (keys and queries.)

(a) (1 points) First, consider if we didn't have key-query-value attention. For keys, queries, and values we'll just use x_i; that is, $v_i = q_i = k_i = x_i$. We'll consider a specific set of x_i. In particular, let u_a, u_b, u_c, u_d be mutually orthogonal vectors in $\mathbb{R}^d$, each with equal norm $\|u_a\| = \|u_b\| = \|u_c\| = \|u_d\| = \beta$, where β is very large. Now, let our x_i be:

$$x_1 = u_d + u_b \tag{9}$$
$$x_2 = u_a \tag{10}$$
$$x_3 = u_c + u_b \tag{11}$$

If we perform self-attention with these vectors, what vector does c_2 approximate? Would it be possible for c_2 to approximate u_b by adding either u_d or u_c to x_2? Explain why or why not (either math or English is fine).

(b) (1 points) Now consider using key-query-value attention as we've defined it originally. Using the same definitions of x_1, x_2 and x_3 as in part i, specify matrices K, Q, V such that $c_2 \approx u_b$, and $c_1 \approx u_b - u_c$. There are many solutions to this problem, so it will be easier for you (and the graders), if you first find V such that $v_1 = u_b$ and $v_3 = u_b - u_c$, then work on Q and K. Some outer product properties may be helpful (as summarized in this footnote)[4].

---

[4]For orthogonal vectors $u, v, w \in \mathbb{R}^d$, the outer product $uv^\top$ is a matrix in $\mathbb{R}^{d \times d}$, and $(uv^\top)v = u(v^\top v) = u\|v\|_2^2$, and $(uv^\top)w = u(v^\top w) = u * 0$. (The last equality is because v and w are orthogonal.)

This handout includes space for every question that requires a written response. Please feel free to use it to handwrite your solutions (legibly, please). If you choose to typeset your solutions, the `README.md` for this assignment includes instructions to regenerate this handout with your typeset LATEX solutions.

References

[1] RADFORD, A., NARASIMHAN, K., SALIMANS, T., AND SUTSKEVER, I. Improving language understanding with unsupervised learning. *Technical report, OpenAI* (2018).

[2] RAFFEL, C., SHAZEER, N., ROBERTS, A., LEE, K., NARANG, S., MATENA, M., ZHOU, Y., LI, W., AND LIU, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research 21*, 140 (2020), 1–67.

[3] TAY, Y., BAHRI, D., METZLER, D., JUAN, D.-C., ZHAO, Z., AND ZHENG, C. Synthesizer: Rethinking self-attention in transformer models. *arXiv preprint arXiv:2005.00743* (2020).