ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Large Language Models
Spring 2023

Profs. Mrinmaya Sachan and Florian Tramèr

Assignment 2

14/08/2023 - 08:46h

You can obtain at most 100 points in this assignment. The grading scheme is the following:

Final assignment grade	Total number of points
6.0	$[96, 100]$
5.75	$[92, 96)$
5.5	$[86, 92)$
5.25	$[82, 86)$
5.0	$[72, 82)$
4.75	$[64, 72)$
4.5	$[58, 64)$
4.25	$[50, 58)$
4.0	$[40, 50)$

How to submit: You need to submit a single zip file titled `firstname_lastname.zip`. The zip file should contain all the required files:

1. All the contents of the latex folder. You just need to enter solutions to the theoretical questions by filling in the "solution" macro after each question.
2. For the programming questions, the required files are mentioned in the various questions.

Question 1: Parameter-Efficient Transfer Learning [Wangchunshu] (20 pts)

Background: Transformer models are composed of L stacked blocks, where each block contains two types of sub-layers: multi-head self-attention and a fully connected feed-forward network (FFN).[1] The conventional attention function maps queries $\boldsymbol{Q} \in \mathbb{R}^{n \times d_k}$ and key-value pairs $\boldsymbol{K} \in \mathbb{R}^{m \times d_k}, \boldsymbol{V} \in \mathbb{R}^{m \times d_v}$:

$$\text{Attn}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d_k}})\boldsymbol{V}, \tag{1}$$

where n and m are the number of queries and key-value pairs respectively. Multi-head attention performs the attention function in parallel over N_h heads, where each head is separately parameterized by $\boldsymbol{W}_q^{(i)}, \boldsymbol{W}_k^{(i)}, \boldsymbol{W}_v^{(i)} \in \mathbb{R}^{d \times d_h}$ to project inputs to queries, keys, and values. Given a sequence of m vectors $\boldsymbol{C} \in \mathbb{R}^{m \times d}$ over which we would like to perform attention and a query vector $\boldsymbol{x} \in \mathbb{R}^d$, multi-head attention (MHA) computes the output on each head and concatenates them:[2]

$$\text{MHA}(\boldsymbol{C}, \boldsymbol{x}) = \text{Concat}(\text{head}_1, \cdots, \text{head}_\text{h})\boldsymbol{W}_o, \ \text{head}_\text{i} = \text{Attn}(\boldsymbol{x}\boldsymbol{W}_q^{(i)}, \boldsymbol{C}\boldsymbol{W}_k^{(i)}, \boldsymbol{C}\boldsymbol{W}_v^{(i)}), \tag{2}$$

where $\boldsymbol{W}_o \in \mathbb{R}^{d \times d}$. d is the model dimension, and in MHA d_h is typically set to d/N_h to save parameters, which indicates that each attention head is operating on a lower-dimensional space. The other important sublayer is the fully connected feed-forward network (FFN) which consists of two linear transformations with a ReLU activation function in between:

$$\text{FFN}(\boldsymbol{x}) = \text{ReLU}(\boldsymbol{x}\boldsymbol{W}_1 + \boldsymbol{b}_1)\boldsymbol{W}_2 + \boldsymbol{b}_2, \tag{3}$$

where $\boldsymbol{W}_1 \in \mathbb{R}^{d \times d_m}$, $\boldsymbol{W}_2 \in \mathbb{R}^{d_m \times d}$. Transformers typically use a large d_m, e.g. $d_m = 4d$. Finally, a residual connection is used followed by layer normalization [1].

a) **(3 pts)** Describe the architecture of the adapter network described in [3] and explain how adapter modules can be added to a pre-trained language model and fine-tuned on downstream tasks. Also, please explain how to calculate the number of trainable parameters when fine-tuning a Transformer encoder (or decoder) with N layers with a hidden dimension of d using adapters of dimension m.

The adapter network consists of a two layer feed forward neural network, with a bottleneck of dimension m. The input of dimension d is projected down to the bottleneck with the first layer, then a nonlinearity is applied. Consecutively the output is projected back to the original dimension with the second layer. Further the adapter network features a skip connection, which connects the input to the output. With this, the adapter acts approximately as an identity function, when its weights are initialized near zero.

Adapter modules can be added after each multi-head attention sublayer and feed-forward sublayer in a Transformer model. During training for a specific task, only the adapters, layer norm parameters and the parameters of the task-specific layers are updated.

For each of the two layers in the adapter network, we need $m \times d$ parameters for the weights. For the biases we need m parameters for the first layer and d parameters for the second layer. That makes $2md + m + d$ parameters for each individual adapter. Additionally we need $2d$ parameters for each layer norm. As there are 2 adapters and 2 layer norms per Transformer layer, we have

$$2N(2md + m + d) + 2N \times 2d = 4Nmd + 2Nm + 6Nd$$

trainable parameters in total. Further there are still some trainable parameters in the task-specific layers.

b) **(3 pts)** Write down the equation of the part where the computation graph is modified by the adapter. (please re-use the notations from the background section whenever possible).

[1] In an encoder-decoder architecture, the transformer decoder usually has another multi-head cross-attention module between the self-attention and FFN, which we omit here for simplicity.

[2] Below, we sometimes ignore the head index i to simplify notation when there is no confusion.

After that, try to convert the equation into the form of:

$$\boldsymbol{h} \leftarrow \alpha \boldsymbol{h} + \lambda \Delta \boldsymbol{h}, \tag{4}$$

and identify what $\Delta \boldsymbol{h}$ is. Most of the times, α and λ are 1 or an independent constant, but when they are not (e.g., there's some relation between them), please specify.

The adapters are inserted after each multi-head attention sublayer and feed-forward sublayer in a Transformer model. Given the output $\boldsymbol{h}$ of a sublayer, the adapter computes,

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + f(\boldsymbol{h}\boldsymbol{W}_{\text{down}} + \boldsymbol{b}_{\text{down}})\boldsymbol{W}_{\text{up}} + \boldsymbol{b}_{\text{up}},$$

where $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d \times m}, \boldsymbol{W}_{\text{up}} \in \mathbb{R}^{m \times d}, \boldsymbol{b}_{\text{down}} \in \mathbb{R}^{m}, \boldsymbol{b}_{\text{up}} \in \mathbb{R}^{d}$ and f is a non-linear activation function. Consequently, it follows that, $\Delta \boldsymbol{h} = f(\boldsymbol{h}\boldsymbol{W}_{\text{down}} + \boldsymbol{b}_{\text{down}})\boldsymbol{W}_{\text{up}} + \boldsymbol{b}_{\text{up}}$.

c) **(3 pts)** In the class notes, we have an additinal reading on LoRA [4]. Describe the general idea of LoRA and how LoRA can be added to a pre-trained language model and fine-tuned on downstream tasks. Also, please explain how to calculate the number of trainable parameters when fine-tuning a Transformer encoder (or decoder) with N layers, H attention heads in each layer, and weight matrices $\boldsymbol{W}_{\text{down}} \in \mathbb{R}^{d \times r}$ using LoRA of bottleneck dimension b.

For a given pre-trained weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times k}$, LoRA seeks to approximate the change $\Delta \boldsymbol{W}$, which is necessary to adapt the weights to a new task. The key idea being, that pre-trained language models have a low "intrinsic dimension" and hence $\Delta \boldsymbol{W}$ can be approximated by a low rank decomposition such that $\Delta \boldsymbol{W} = BA$, where $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$ and $r \ll \min(d, k)$. During adaption to a new task, for each weight matrix $\boldsymbol{W}$ only B and A are trained. In contrast to adapter modules, LoRA does not introduce additional inference overhead, as the learned change $\Delta \boldsymbol{W}$ can be applied directly to the pre-trained weights $\boldsymbol{W}$.

In the paper, LoRA is applied to the matrices $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v \in \mathbb{R}^{d \times d_h}$ of each attention head and further to $\boldsymbol{W}_o \in \mathbb{R}^{d \times d}$. That means we get $3(dr + rd_h)$ parameters for a single attention head. $\boldsymbol{W}_o$ adds a another $2dr$ parameters per layer. Finally, for all N layers and H attention heads per layer, we thus have

$$N(3H(dr + rd_h) + 2dr)$$

trainable parameters in total.

d) **(3 pts)** Write down the equation of the part where the computation graph is modified by LoRA. (please re-use the notations from the background section whenever possible).

After that, try to convert the equation into the form of:

$$\boldsymbol{h} \leftarrow \alpha \boldsymbol{h} + \lambda \Delta \boldsymbol{h}, \tag{5}$$

and identify what $\Delta \boldsymbol{h}$ is. Most of the time, α and λ are 1 or an independent constant, but when they are not (e.g., there's some relation between them), please specify.

Given an input $\boldsymbol{x} \in \mathbb{R}^{d}$ and a weight matrix $\boldsymbol{W} \in \mathbb{R}^{d \times k}$, LoRA modifies the computation graph as follows,

$$\boldsymbol{h} \leftarrow \boldsymbol{h} + \frac{\alpha}{r}\boldsymbol{x}BA,$$

where $\boldsymbol{h} = \boldsymbol{x}\boldsymbol{W}$, $B \in \mathbb{R}^{d \times r}, A \in \mathbb{R}^{r \times k}$. Consequently, it follows that, $\Delta \boldsymbol{h} = \boldsymbol{x}BA$ and $\lambda = \frac{\alpha}{r}$.

e) **(3 pts)** Describe how prefix-tuning can be added to a pre-trained language model and used for fine-tuning on downstream tasks. Also, please explain how to calculate the number of trainable parameters when fine-tuning a Transformer encoder (or decoder) with N layers and H attention heads with dimensionality d using prefix-tuning with l embedding vectors each layer.

In prefix-tuning, a sequence of continuous task specific vectors is prepended to the input of all transformer blocks. During fine-tuning only these vectors are trained and the rest of the pre-trained model is frozen. The idea is, that the Transformer can attend to these prefix vectors and use them as context. These embeddings are continuous and do not have to correspond to tokens, so gradient descent can be used to update them during fine-tuning. We will get Nld trainable parameters as only a fixed part of the input of each Transformer block is changed.

f) **(2 pts)** In [6] (see Sec 4.3 of [6]), the prefix is reparametrized by a smaller matrix composed with an MLP. Assuming the smaller matrix has the same dimensionality as the hidden size of the model, and the MLP has two layers with an intermediate size of d', please again explain how to calculate the number of trainable parameters for the above model.

The MLP takes as input a vector of dimension d and outputs a vector of dimension of $N * d$. Hence we will get two matrices of dimension $d \times d'$ and $d' \times (N * d)$. Additionally we have to account for the bias parameters, which are of size d' and $N * d$ respectively. Further we have to account for the parameters of the smaller matrix, which is of size $l \times d$. That makes in total

$$dd' + d'Nd + d' + Nd + ld$$

trainable parameters.

g) **(3 pts)** Please write down the equation of the part where the computation graph is modified by prefix-tuning. (please re-use the notations from the background section whenever possible).
After that, try to convert the equation into the form of:

$$\boldsymbol{h} \leftarrow \alpha \boldsymbol{h} + \lambda \Delta \boldsymbol{h}, \tag{6}$$

and identify what $\Delta \boldsymbol{h}$ is. Most of the time, α and λ are 1 or an independent constant, but when they are not (e.g., there's some relation between them), please specify.
Hints:

$$\begin{aligned}
&\text{head} = \text{Attn}(\boldsymbol{x}\boldsymbol{W}_q, \text{concat}(\boldsymbol{P}_k, \boldsymbol{C}\boldsymbol{W}_k), \text{concat}(\boldsymbol{P}_v, \boldsymbol{C}\boldsymbol{W}_v)) \\
&= \text{softmax}\big(\boldsymbol{x}\boldsymbol{W}_q \text{concat}(\boldsymbol{P}_k, \boldsymbol{C}\boldsymbol{W}_k)^\top\big) \begin{bmatrix} \boldsymbol{P}_v \\ \boldsymbol{C}\boldsymbol{W}_v \end{bmatrix} \\
&= (1 - \lambda(\boldsymbol{x}))\text{softmax}(\boldsymbol{x}\boldsymbol{W}_q\boldsymbol{W}_k^\top\boldsymbol{C}^\top)\boldsymbol{C}\boldsymbol{W}_v + \lambda(\boldsymbol{x})\text{softmax}(x\boldsymbol{W}_q\boldsymbol{P}_k^\top)\boldsymbol{P}_v \\
&= (1 - \lambda(\boldsymbol{x}))\underbrace{\text{Attn}(\boldsymbol{x}\boldsymbol{W}_q, \boldsymbol{C}\boldsymbol{W}_k, \boldsymbol{C}\boldsymbol{W}_v)}_{\text{standard attention, h}} + \lambda(\boldsymbol{x})\underbrace{\text{Attn}(\boldsymbol{x}\boldsymbol{W}_q, \boldsymbol{P}_k, \boldsymbol{P}_v)}_{\text{independent of } \boldsymbol{C},\ \Delta h},
\end{aligned} \tag{7}$$

where $\lambda(\boldsymbol{x})$ is a scalar that represents the sum of normalized attention weights on the prefixes. Please calculate the expression for $\lambda(\boldsymbol{x})$.

$$\boldsymbol{h} \leftarrow \text{Attn}(\boldsymbol{x}\boldsymbol{W}_q, \text{concat}(\boldsymbol{P}_k, \boldsymbol{C}\boldsymbol{W}_k), \text{concat}(\boldsymbol{P}_v, \boldsymbol{C}\boldsymbol{W}_v)),$$

where $\boldsymbol{P}_k$ are the keys of the prefixes, $\boldsymbol{P}_v$ are the values of the prefixes.
We can write the softmax as,

$$a_j = \text{softmax}(\boldsymbol{x}\boldsymbol{W}_q \text{concat}(\boldsymbol{P}_k, \boldsymbol{C}\boldsymbol{W}_k)^\top)_j = \frac{\exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_j\right)}{\sum_i \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)}$$

where the k_j are the keys of prefixes and inputs. Let $\mathcal{P}$ be the set of indices of the prefixes. In order to split the softmax we can transform the a_j as follows,

$$\begin{aligned}
a_j &= \frac{\sum_{i \in \mathcal{P}} \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)}{\sum_i \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)} \frac{\exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_j\right)}{\sum_{i \in \mathcal{P}} \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)}, \text{ if } j \in \mathcal{P} \\
a_j &= \frac{\sum_{i \notin \mathcal{P}} \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)}{\sum_i \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)} \frac{\exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_j\right)}{\sum_{i \notin \mathcal{P}} \exp\left(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i\right)}, \text{ if } j \notin \mathcal{P}.
\end{aligned}$$

As each softmax component is multiplied with its corresponding value this gives the desired split with,

$$\lambda(\boldsymbol{x}) = \frac{\sum_{i \in \mathcal{P}} \exp(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i)}{\sum_i \exp(\boldsymbol{x}\boldsymbol{W}_q \cdot k_i)}.$$

Question 2: Scaling Laws for Language Models [Alessandro] (15 points)

Note: In order to answer this question, please refer to the class notes. You are also allowed to read the scaling laws paper [5] for this question. Scaling laws are empirical relationships that describe how certain properties of a system change as a function of its size or scale. In neural language models, scaling laws describe how performance on the cross-entropy loss scales as a power-law with model size, dataset size, and the amount of compute used for training.

In particular, the test loss L of a Transformer trained to autoregressively model language can be predicted when performance is limited by only either the number of parameters N or the dataset size D as follows:

$$L(N) = \frac{A}{N^{\alpha}} \; ; \qquad L(D) = \frac{B}{D^{\beta}} \; ; \tag{8}$$

where A and B are some constants.

a) **(4 pts)** Lets say we want to train a language model. Assume that we are at an optimal point in training. In other words, if we were to increase the dataset size D, then the limiting factor would be N, and if we were to increase the parameter number N the D would become the bottleneck. We want to further train the model preserving this optimality, how should we increase the two quantities N and D simultaneously? (You can write how D should scale as a function of N or vice versa.)

At the optimal point we have $L(N) = L(D)$, i.e. both limit the performance equally. Hence we can solve for D,

$$\begin{aligned} \frac{A}{N^{\alpha}} &= \frac{B}{D^{\beta}} \\ D^{\beta} &= \frac{B}{A} N^{\alpha} \\ D &= \left(\frac{B}{A}\right)^{\frac{1}{\beta}} N^{\frac{\alpha}{\beta}}. \end{aligned}$$

b) **(4 pts)** Given a fixed amount of training compute $C = ND$ (number of FLOPs $\propto$ number of parameters $\times$ number of training steps), now use the result from the previous question to compute the portion of C that you would assign to each of the two factors dataset size and parameter count. (Hint: given $x, y \in [0, 1]$ such that $x + y = 1$, $C = C^x C^y$. We would like to know x and y, such that $N = C^x$ and $D = C^y$.)

We have two equations,

$$C^x = N \text{ and } C^{1-x} = D = \left(\frac{B}{A}\right)^{\frac{1}{\beta}} N^{\frac{\alpha}{\beta}}.$$

Plugging in the first equation into the second one yields,

$$\begin{aligned} C^{1-x} &= \left(\frac{B}{A}\right)^{\frac{1}{\beta}} C^{x\frac{\alpha}{\beta}} \\ (1-x)\log C &= \frac{1}{\beta}\log\frac{B}{A} + x\frac{\alpha}{\beta}\log C \\ x(\frac{\alpha}{\beta}+1)\log C &= \log C - \frac{1}{\beta}\log\frac{A}{B} \\ x &= \frac{\log C - \frac{1}{\beta}\log\frac{B}{A}}{(\frac{\alpha}{\beta}+1)\log C}. \end{aligned}$$

Hence, y is given by,

$$y = 1 - x = \frac{\frac{\alpha}{\beta}\log C + \frac{1}{\beta}\log\frac{B}{A}}{(\frac{\alpha}{\beta}+1)\log C}.$$

c) **(4 pts)** A recent study estimated the value of the two coefficients to be $x = 0.46$ and $y = 0.54$ [2]. Consider a model with 100 billion parameters is trained on a corpus consisting of 300 billion tokens, is the model trained optimally? If not, how would you further train it to reach optimality?[3]

We can simply plug in the numbers and compute,

$$C^{0.46} = (ND)^{0.46} = 2.19 * 10^{10} < N = 100 * 10^9 = 10^{11}$$
$$C^{0.54} = (ND)^{0.54} = 1.37 * 10^{12} > D = 300 * 10^9 = 3 * 10^{11}.$$

Since the numbers do not match, we can conclude that the model is not trained optimally. Considering that increasing the amount of compute C is not always possible, we could fix D and then decrease N till the equation $(ND)^{0.46} = N$ is satisfied. Then the training would be optimal. Precisely we would need to decrease N to,

$$(N_{\text{opt}}D)^{0.46} = N_{\text{opt}}$$
$$D^{0.46} = N_{\text{opt}}^{0.54}$$
$$N_{\text{opt}} = D^{\frac{0.46}{0.54}} = 5.98 * 10^9.$$

d) **(3 pts)** Besides the power-law functional form, what are some other assumptions that we incorporated in Eq. 8?

- We assume that the performance is limited by N and D independently, which may not be the case.
- The given power laws also imply that the loss will get very small for large N and D. But it might be the case that the loss saturates at some point. As mentioned in the lecture, datasets start to suffer in quality when they get very large.
- The loss depends on training time as well, which naturally grows with bigger datasets and more parameters. This quantity is not considered in the given power laws.
- There is no distinction between different types of parameters, i.e. embedding parameters and model parameters. Although they might impact the loss differently.

[3] Our setting introduces multiple additional assumptions compared to the study in Hoffman et al. [2], so our results will deviate from the ones presented in the paper.

Question 3: Prompting Language Models [Shridhar] (15 pts + 2.5 (Bonus))

Overview: Modern prompting techniques with LLMs involve strategically designing input prompts to guide the model's behavior and improve its performance on specific tasks. These techniques have become increasingly important as LLMs demonstrate remarkable capabilities when given appropriate prompts. In this question, we will try to explore the effect of these prompting variants for various language models (particularly, T5 and Flan-T5).

In particular, in this question, we will explore the domain of math word problems which require a combination of text understanding and arithmetic reasoning. We will test cases when training data vs. no training data is available for solving the task of solving math word problems. In a practical setting, we will understand the usefulness of concepts like in-context learning, explicit instructions, instruction-tuned models, and step-by-step guidance.

Instructions: The set of questions and our implementation pipeline is provided in the Google Colab Notebook. All task-related dependencies and step-by-step instructions are also provided in the Colab notebook. It is strongly recommended to use Google Colab for running all the experiments (except question 1 for which Euler GPUs will be needed) and download the Colab file in a `.ipynb` format and submit the file in the following naming: `Q3.ipynb`

Question 4: Watermarking LLM outputs (25 pts)

Overview. In this question, you will implement various approaches for watermarking LLM outputs (see the lecture on LLM Security for a short introduction to watermarking).

Specifically, you will implement three watermarking schemes:

1. A dummy scheme that generates text that never contains any letter 'e' (lowercase or uppercase).
2. A red-list scheme that proceeds as follows: when generating a token from the LLM, use the value of the previously generated token to seed a PRNG, and use this PRNG to randomly split the set of tokens into a "red list" and a "green list". Then, only output a token from the green list.
3. The above scheme can be too strict. E.g., suppose you just generated the token 'Barack', and the token 'Obama' ends up in the red list. Then your LLM can never generate the string 'Barack Obama'. To alleviate this, implement a soft version of the red-list scheme: instead of outright banning the LLM from outputting tokens from the red list, we will simply bias the model against these tokens, by reducing the logit scores of all tokens in the red list by some value γ.

For 20 different text prompts that you are given, you should then generate 100 tokens with the LLM using each of these watermarking schemes.

Once you have the watermarking schemes working, think about how you would go about detecting these watermarks.

For this, we will give you 80 pieces of text, that have been generated either with: (1) no watermarking; (2) the dummy "no e" watermark; (3) a red list watermark; (4) a soft red list watermark. You should then figure out which piece of text was produced with which type of watermarking.

Instructions. You will work with the following Google Colab Notebook.

We will be using the GPT-2 LLM. While this model is a bit dated (and thus does not produce text as fluent as more modern LLMs), it is small enough to run easily on the GPUs you can get for free on Colab!

The Colab Notebook contains all the instructions to get you started. In particular, we provide you with a default configuration for text generation, as well as simple PRNG implementation to convert tokens to red lists.

Submission. For this question, you will submit your solutions as two `NumPy` arrays.

1. `Q4_gens.npy`, is an array of 60 strings that contains your watermarked generations (3 different watermarking schemes applied to 20 different prompts).
2. `Q4_guesses.npy`, is an array of integers of size 80 that contains your guess of which watermarking scheme was used for each of the 80 provided text pieces (1 = no watermarking; 2 = the dummy "no e" watermark; 3 = a red list watermark; 4 = a soft red list watermark). **(each of the 4 options was used exactly 20 times).**

Question 5: Secretly censoring Stable Diffusion with LLM embeddings (25 pts)

Overview. You've probably heard of (and maybe used) image generation AIs like Stable Diffusion, Midjourney or DALL-E.

These systems take in a "prompt" (a string like "A picture of a cat") and then generate an image that best matches the prompt.

Due to concerns that people would use their system to generate bad content, Stable Diffusion added a *Safety filter*, which works as follows:

- When an image x is generated by the system, it is fed through an *image encoder* (based on OpenAI's CLIP model) that generates an *embedding vector* $z \leftarrow \mathsf{encode}_{\text{img}}(x) \in \mathbb{R}^{1024}$.
- This embedding is then compared to a list of fixed "bad embeddings". If the cosine similarity between the image's embedding z and one of these bad embeddings is above some threshold, the image is considered bad and discarded.
- The "bad embeddings" are generated by encoding *text* that corresponds to bad things (e.g., pornography, violence, etc.) This works because the CLIP model used for encoding images is actually a *multimodal* model, than can encode both images and text, with the property that text and images that represent similar things should get encoded to similar embeddings. Concretely, the bad embeddings are computed as $\mathsf{encode}_{\text{text}}(s) \in \mathbb{R}^{1024}$ for some text s.

For some "extra security", Stable Diffusion did not release the banned text strings s. Instead, they only provide the list of bad embeddings. Your goal is to recover these banned strings.

To make sure you don't have to work with actual NSFW content, we have created a set of 17 bad embeddings that correspond to fairly benign things we don't want you to be able to generate images of (one of these bad embeddings is an encoding of the string "The solutions to this assignment", to make sure you don't use Stable Diffusion to try and cheat).

So your actual goal will be to recover the pieces of text that we encoded to generate these bad embeddings.

Instructions. You will work with the following Google Colab Notebook.

The Colab Notebook contains all the instructions to get you started, and to download the data needed for this question.

Submission. Submit your 15 guesses in a TXT file named `Q5.txt`, with one guess per line. The Colab Notebook contains a code snippet to generate this file for you.

References

[1] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016.

[2] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[3] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *Proceedings of Machine Learning Research*, pages 2790–2799. PMLR, 09–15 Jun 2019.

[4] Edward Hu, Yelong Shen, Phil Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models, 2021.

[5] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.

[6] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, Online, August 2021. Association for Computational Linguistics.

A Using Google Colab

Here's a simple workflow to get you started with Colab:

https://scribehow.com/shared/Colab_Workflow_Short__-_n5wpWVSOyizxH1i1GxOg

Colab gives you access to one GPU "for free", but depending on your usage and on resource availability, the service may decide not to grant you a GPU. We thus recommend the following:

- While you're familiarizing yourself with the codebase, or thinking about how to solve the problems, disconnect the GPU runtime.
- If you do run out of GPU resources on Colab, there is a backup solution using the Euler cluster (see below).

B Notebooks, Pytorch, NumPy, GPUs

If you have any questions or run into any issues, ask us on RocketChat. You can also find good explanations for the things you need to know for the lab (and a lot more) here:

- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_numpy.ipynb
- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_pytorch.ipynb
- https://nbviewer.org/github/cgpotts/cs224u/blob/2020-spring/tutorial_jupyter_notebooks.ipynb

C Running Jupyter Notebooks on Euler

If you run out of free GPUs on Colab, there is an option to use a GPU running on the Euler cluster as an alternative runtime. The process is a bit more involved, and GPU resources on Euler are limited, so please only do this if you do run out of Colab's free tier.

C.1 Launch a Jupyter Notebook from Euler

1. Connect to Euler (see instructions).
2. download the `start_jupyter.sh` script:

```
$ curl -O https://gist.githubusercontent.com/ftramer/b519b4e3d204189b27f94b058a6c4d20/raw/
    start_jupyter.sh
```

3. Start a `Jupyter` notebook server on an Euler node with a GPU:

```
$ sbatch start_jupyter.sh
```

   This should print out the message: "`Submitted batch job {JOB_ID}`".

4. You can use the command `squeue` to check if your job is running. You will also get an email once it starts.

   Once it is running, your job will print out information to two local files, `jupyter.out` and `jupyter.err`. The file `jupyter.out` should contain something like this:

```
Run the following command on your local machine to enable port forwarding:
ssh -N -L 8888:{NODE_IP}:8888 {USER}@login.euler.ethz.ch
```

5. Run the command in `jupyter.out` on your **local machine**.

```
$ ssh -N -L 8888:{NODE_IP}:8888 {USER}@login.euler.ethz.ch
```

6. Now open the file `jupyter.err` and copy the URL that is printed at the bottom (this may take a minute to appear). It should look like this:

```
http://127.0.0.1:8888/lab?token=d70a2ae9...
```

7. Open this URL in the browser on your local machine. You should now be able to see the `Jupyter` notebook interface.

C.2 Shutting down the Jupyter Notebook

Run `squeue` on Euler to get the JOB_ID of your job. Then run: `scancel JOB_ID`. Finally, kill the SSH tunnel on your local machine by pressing `Ctrl+C`.