EECS 182 Deep Neural Networks
Fall 2023 Anant Sahai

Homework 11

This homework is due on November 18, at 10:59PM.

1. Pretraining and Finetuning

(a) Consider the following two pretrained language models: BERT and GPT-2. We would like to fine-tune those models for the following four tasks: sentimental analysis, summarization, named entity recognition, and translation. **Which fine-tuned model will perform best for each task?**

(b) When we use a pretrained model without fine-tuning, we typically just train a new task-specific head. With standard fine-tuning, we also allow the model weights to be adapted.

However, it has recently been found that (Lee, Yoonho, et al, 2022)[1] we can selectively fine-tune a subset of layers to get better performance especially under certain kinds of distribution shifts on the inputs. Suppose that we have a ResNet-26 model pretrained with CIFAR-10. Our target task is CIFAR-10-C, which adds pixel-level corruptions (like adding noise, different kinds of blurring, pixelation, changing brightness and contrast, etc) to CIFAR-10. Examples are in Figure 1.

If we could only afford to fine-tune one layer, **which layer (i.e. 1,2,3,4,5) in Figure 2 should we choose to finetune to get the best performance on CIFAR-10-C? Give brief intuition as to why.**

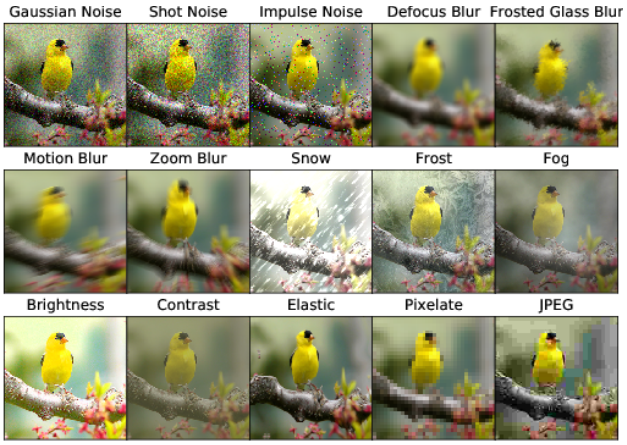

Figure 1: The ImageNet-C dataset, obtained from corrupting ImageNet. The CIFAR-10-C dataset is similarly obtained by corrupting CIFAR-10 with similar corruption methods. Figure from Corruption and Perturbation Robustness (ICLR 2019).

[1] Lee, Yoonho, et al. "Surgical fine-tuning improves adaptation to distribution shifts." arXiv preprint arXiv:2210.11466 (2022).

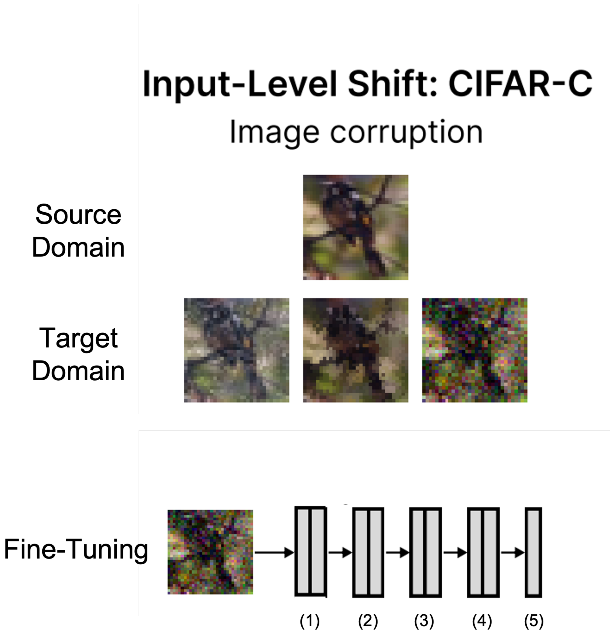

Figure 2: Fine-tuning the model pretrained with CIFAR-10 on CIFAR-10-C dataset. Figure 1 of (Lee, Yoonho, et al, 2022).

2. Prompting Language Models

(a) **Exploring Pretrained LMs**

Play around with the web interface at `https://dashboard.cohere.ai/playground/generate`. This plaground provides you an interface to interact with a large language model from Cohere and tweak various parameters. You will need to sign up for a free account.

Once you're logged in, you can choose a model in the parameters pane on the right. `command-xlarge-nightly` is a generative model that responds well with instruction-like prompts. `xlarge` and `medium` are generative models focusing on sentence completion. Spend a while exploring prompting these models for different tasks. Here are some suggestions:

- Look through the 'Examples . . . ' button at the top of the page for example prompts.
- Ask the model to answer factual questions.
- Prompt the model to generate a list of 100 numbers sampled uniformly between 0 and 9. Are the numbers actually randomly distributed?
- Insert a poorly written sentence, and have the model correct the errors.
- Have the model brainstorm creative ideas (names for a storybook character, recipes, solutions to solve a problem, etc.)
- Chat with the model like a chatbot.

Answer the questions below:

i. **Describe one new thing you learned by playing with these models.**

ii. **How does the temperature parameter affect the outputs?** Justify your answer with a few examples.

iii. **Describe a task where the larger models (e.g., `xlarge` or `command-xlarge-nightly`) significantly outperform the smaller ones (e.g., `medium`).** Paste in examples from the biggest and smallest model to show this.

iv. **Describe a task where even the largest model performs badly**. Paste in an example to show this.

v. **Describe a task where the model's outputs improve significantly with few-shot prompting compared to zero-shot prompting.**

(b) **Using LMs for classification**

Run `lm_prompting.ipynb`, then answer the following questions. If you did not do part (a), you will still need to get a Cohere account to complete this part.

i. Analyze the command-xlarge-nightly model's failures. **What kinds of failures do you see with different prompting strategies?**

ii. **Does providing correct labels in few-shot prompting have a significant impact on accuracy?**

iii. **Observe the model's log probabilities. Does it seem more confident when it is correct than when it is incorrect?**

iv. **Why do you think the GPT2 model performed so much worse than the command-xlarge-nightly model on the question answering task?**

v. **How did soft prompting compare to hard prompting on the pluralize task?**

vi. You should see that when the model fails (especially early in training of a soft prompt or with a bad hard prompt) it often outputs common but uninformative tokens such as the, ", or \n. **Why does this occur?**

3. Soft-Prompting Language Models

You are using a pretrained language model with prompting to answer math word problems. You are using chain-of-thought reasoning, a technique that induces the model to "show its work" before outputting a final answer.

Here is an example of how this works:

```
[prompt] Question: If you split a dozen apples evenly among yourself
and three friends, how many apples do you get? Answer: There are 12
apples, and the number of people is 3 + 1 = 4. Therefore, 12 / 4 = 3.
Final answer: 3\n
```

If we were doing hard prompting with a frozen language model, we would use a hand-designed [prompt] that is a set of tokens prepended to each question (for instance, the prompt might contain instructions for the task). At test time, you would pass the model the sequence and end after "Answer:" The language model will continue the sequence. You extract answers from the output sequence by parsing any tokens between the phrase "Final answer: " and the newline character "\n".

(a) Let's say you want to improve a frozen GPT model's performance on this task through soft prompting and training the soft prompt using a gradient-based method. This soft prompt consists of 5 vectors prepended to the sequence at the input — these bypass the standard layer of embedding tokens into vectors. (Note: we do not apply a soft prompt at other layers.) Imagine an input training sequence which looks like this:

```
["Tokens" 1-5: soft prompt] [Tokens 6-50: question]
[Tokens 51-70: chain of thought reasoning]
[Token 71: answer] [Token 72: newline]
[Tokens 73-100: padding].
```

We compute the loss by passing this sequence through a transformer model and computing the cross-entropy loss on the output predictions. If we want to train the soft-prompt to output correct reasoning and produce the correct answer, **which output tokens will be used to compute the loss?** (Remember that the target sequence is shifted over by 1 compared to the input sequence. So, for example, the answer token is position 71 in the input and position 70 in the target).

(b) Continuing the setup above, **how many parameters are being trained in this model?** You may write this in terms of the max sequence length S, the token embedding dimension E, the vocab size V, the hidden state size H, the number of layers L, and the attention query/key feature dimension D.

(c) **Mark each of the following statements as True or False and give a brief explanation.**

(i) If you are using an autoregressive GPT model as described in part (a), it's possible to precompute the representations at each layer for the indices corresponding to prompt tokens (i.e. compute them once for use in all different training points within a batch).

(ii) If you compare the validation-set performance of the *best possible* K-token hard prompt to the *best possible* K-vector soft prompt, the soft-prompt performance will always be equal or better.

(iii) If you are not constrained by computational cost, then fully finetuning the language model is always guaranteed to be a better choice than soft prompt tuning.

(iv) If you use a dataset of samples from Task A to do prompt tuning to generate a soft prompt which is only prepended to inputs of Task A, then performance on some other Task B with its own soft prompt might decrease due to catastrophic forgetting.

(d) Suppose that you had a family of related tasks for which you want use a frozen GPT-style language model together with learned soft-prompting to give solutions for the task. Suppose that you have substantial training data for many examples of tasks from this family. **Describe how you would adapt a meta-learning approach like MAML for this situation?**

(HINT: This is a relatively open-ended question, but you need to think about what it is that you want to learn during meta-learning, how you will learn it, and how you will use what you have learned when faced with a previously unseen task from this family.)

4. Meta-learning for Learning 1D functions

Preliminary discussion: Meta-learning and the MAML algorithm.

In meta-learning, there are two learning stages. The fast stage, or the learning episode, is like a single organism born to play a single game of life. The slow stage is like evolution. Correspondingly, there are two kinds of learned parameters: fast weights and slow weights. The fast weights are initialized anew at the birth of each organism, modified during its life, and killed off at the end of its life. The slow weights are initialized at the start of the species, modified at the death of an individual, and never discarded.

During each episode, a learner β_0 is born anew, according to its genome (slow weights) c. Nature samples a task T for the learner. The learner meets one data point after another $(x_1, y_1), (x_2, y_2), ..., (x_H, y_H)$, making updates $\beta_0 \mapsto \beta_1 \mapsto \beta_2 \mapsto \cdots \mapsto \beta_H$. Here the letter H should be read as "horizon".

At the end of life, the learner is $\hat{\beta} := \beta_H$, and a final loss is computed for the learner's life: $\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}_T, \mathbf{c}\right)$. Now the learner is destroyed, and the genome c is updated by backpropagating across the learner's entire life, right back to its birth.[2]

[2] Of course, real biological evolution does not backprop into the genome, but uses natural selection, but the analogy should help you understand I hope. Also, "life flashing before you eyes" is not supposed to be backpropagation...

The hope of meta-learning is that, if c is updated appropriately, it would eventually end up minimizing the final loss, averaged over all possible tasks:

$$\underset{\mathbf{c}}{\operatorname{argmin}} \mathbb{E}_{\mathcal{D}_T}\left[\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}, \mathbf{c}\right)\right]$$

This question explores meta-learning on a very simple case: meta-learning a neural network that learns a 1D function. The variables involved are as follows:

- c: **slow weights**, modified at the end of each learning episodes.
- β: **fast weights**, modified within a learning episode.
- $\hat{\beta}$: fast weights at the end of a learning episode.
- T: the **task** faced by a learner.
- $\mathcal{D}_T$: **task distribution**, a probability distribution over all possible tasks.
- $\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}, \mathbf{c}\right)$: **final loss** achieved by the learner. Meta-learning should minimize the expectation of this number.

There are many machine learning techniques which can fall under the nebulous heading of meta-learning, but we will focus on one with Berkeley roots called Model Agnostic Meta-Learning (MAML)[3] which optimizes the initial weights of a network to rapidly converge to low loss within the task distribution. The MAML algorithm as described by the original paper is shown in Fig. 3.

```
Algorithm 1 Model-Agnostic Meta-Learning
Require: $p(\mathcal{T})$: distribution over tasks
Require: $\alpha$, $\beta$: step size hyperparameters
1: randomly initialize $\theta$
2: while not done do
3:     Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4:     for all $\mathcal{T}_i$ do
5:         Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to $K$ examples
6:         Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7:     end for
8:     Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: end while
```

Figure 3: MAML algorithm. We will refer to the training steps on line 6 as the *inner update*, and the training step on line 8 as the *meta update*.

At a high level, MAML works by sampling a "mini-batch" of tasks $\{T_i\}$ and using regular gradient descent updates to find a new set of parameters θ_i for each task starting from the same initialization θ. Then the gradient w.r.t. the original θ each calculated for each task using the task-specific updated weights θ_i, and θ is updated with these 'meta' gradients. Fig. 4 illustrates the path the weights take with these updates.

[3] C. Finn, P. Abbeel, S. Levine, "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks," in *Proceedings of the 34th International Conference on Machine Learning, Sydney, Australia, PMLR 70, 2017*

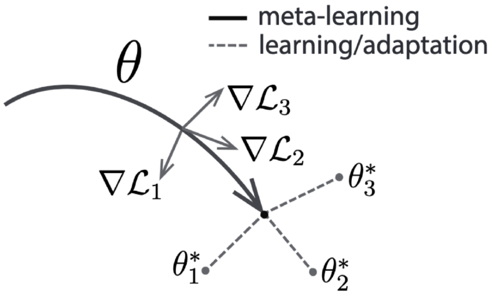

Figure 4: MAML gradient trajectory illustration

The end goal is to produce weights θ^* which can reach a state useful for a particular task from $\mathcal{D}_T$ after a few steps — needing to use less data to learn. If you want to understand the fine details of the algorithm and implementation, we recommend reading the original paper and diving into the code provided with this problem.

Let's make a tractable toy problem, by adding severe and unrealistic simplifications.

In this problem we consider functions of type $\mathbb{R} \to \mathbb{R}$. The space of all possible $\mathbb{R} \to \mathbb{R}$ functions is too vast to consider, so we only take a finite-dimensional subspace of it. Specifically, we pick d functions $\phi_0, \phi_1, ..., \phi_{d-1}$, and only allow linear sums of them.

A task is defined by a parameter α: the learner should learn the following function:

$$f_\alpha := \sum_{k=0}^{d-1} \alpha_k \phi_k(x).$$

α is sampled from the probability distribution $\mathcal{D}_T$.

In the original MAML algorithm, the inner loop performs gradient descent to optimize loss with respect to a task distribution. However, a learner that learns by gradient descent makes it intractable to analyze on paper, so we consider a simpler learner: The learner is born, and immediately confronted with a list of datapoints $(x_1, f_\alpha(x_1)), ..., (x_H, f_\alpha(x_H))$. Here, the numbers $x_1, ..., x_H$ are not sampled, but *fixed*! Why? Well, it makes the calculation easier, even though it is quite unrealistic. In this way, the task distribution is purely about sampling a parameter α, and not at all about sampling the training dataset $(x_1, f_\alpha(x_1)), ..., (x_H, f_\alpha(x_H))$.

The learner performs minimal-norm regression on them to obtain $\hat{\beta}$ in one step[4]. The learner is defined by:

$$f_{\hat{\beta},c} := \sum_{k=0}^{d-1} \hat{\beta}_k c_k \phi_k(x).$$

where $\hat{\beta}$ is obtained by minimal norm regression:

$$\hat{\boldsymbol{\beta}} = \underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{\beta}\|_2^2 \tag{1}$$

[4] Least-squares regression is performed in a single step, so we can imagine that the learner has the shortest possible life: it came, it learned, it died.

$$\text{s.t. } y_i = \sum_{k=0}^{d-1} \beta_k c_k \phi_k(x_i) \quad \forall i = 1, 2, ..., K. \tag{2}$$

Althought it appears like $\hat{\beta}$ and c are playing the same role, they do not. Suppose we put c_0 very small, then it would cost a lot of norm on $\hat{\beta}$ to make $\hat{\beta}_0 c_0$ large, and so the learned function would only have a little bit of ϕ_0. Conversely, if c_0 is very large, then the learned function would have a lot of ϕ_0. In this way, c directs the learner to use more or less of each $\phi_0, ..., \phi_{d-1}$, exactly what meta-learning is supposed to do.

Since we want to use min-norm regression, we require that $H \leq d$, so that the min-norm regression always has a solution. Also, recall that the solution to

$$\underset{\boldsymbol{\beta}}{\operatorname{argmin}} \|\boldsymbol{\beta}\| \text{ , such that } \mathbf{y} = A\boldsymbol{\beta}$$

in terms of A and $\mathbf{y}$ is

$$\hat{\boldsymbol{\beta}} = A^\top (AA^\top)^{-1} \mathbf{y}$$

The learner is tested on a single test point $(x_{test}, f_\alpha(x_{test}))$, where x_{test} is sampled from a certain probability distribution μ_{test}. Loss is just the square loss:

$$\mathcal{L}_T\left(\hat{\boldsymbol{\beta}}, \mathbf{c}\right) := \frac{1}{2}(f_\alpha(x_{test}) - f_{\hat{\beta},c}(x_{test}))^2$$

We also assume that the functions ϕ are actually orthonormal with respect to the test distribution. That is,

$$E_{x_{test} \sim \mu_{test}}[\phi_k(x_{test})\phi_l(x_{test})] = \delta_{kl}$$

Compared to the previous assumption that $x_1, ..., x_H$ are fixed, this assumption is actually realistic. Why? Well, if you recall your Fourier analysis, then you would know that if x_{test} is sampled uniformly from $[0, 2\pi]$, and the functions ϕ are trigonometric functions with period $2\pi/N$ for integer N, then they do satisfy the requirement. Similarly, if you sample x_{test} from the standard normal distribution, then the Hermite polynomials work.

In general, meta-learning should work for any $\mathcal{D}_T$. However, to make it easy to eyeball how well our meta-learning is doing, we inflict another simplification. We assume that any α sampled from $\mathcal{D}_T$ can only have nonzero entries on a subset S of $0 : (d-1)$. Then, meta-learning should gradually push c_i to zero for any i not in S.

The set S is hidden from the meta-learner and the learner (no cheating!), but we can judge how well the meta-learning is going by comparing the learner against another learner who can cheat – that is, consult an oracle that will tell it exactly what S is.

In other words, the oracle-consulting learner performs regression using only the features present in the data. We expect to see the meta-learning to gradually create learners that reach the same loss as the oracle-consulting learner.

For the following three problems, we perform pen-and-paper analysis, so we have to inflict *even more simplifications* to make the algebra easy.

(a) Suppose that we have exactly one training point (x, y), and one true feature $\phi_t(x) = \phi_1(x)$. We have a second (alias) feature that is identical to the first true feature, $\phi_a(x) = \phi_2(x) = \phi_1(x)$. This is a caricature of what always happens when we have fewer training points than model parameters.

The function we wish to learn is $y = \phi_t(x)$. We learn coefficients $\hat{\boldsymbol{\beta}}$ using the training data. Note, both the coefficients and the feature weights are 2-d vectors.

Show that $\hat{\boldsymbol{\beta}} = \frac{1}{c_0^2+c_1^2}\begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$

(b) Assume for simplicity that we have access to infinite data from the test distribution for the purpose of updating the feature weights $\mathbf{c}$. **Calculate the gradient of the expected test error with respect to the feature weights c_0 and c_1, respectively:**

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{c}}\left(\mathbb{E}_{x_{test},y_{test}}\left[\frac{1}{2}\left\|y-\hat{\beta}_0c_0\phi_t(x)-\hat{\beta}_1c_1\phi_a(x)\right\|_2^2\right]\right).$$

Use the values for $\boldsymbol{\beta}$ from the previous part. *(Hint: the features $\phi_i(x)$ are orthonormal under the test distribution. They are not identical here.)*

(c) **Generate a plot** showing that, for some initialization $\mathbf{c}^{(0)}$, as the number of iterations $i \to \infty$ the weights empirically converge to $c_0 = \|\mathbf{c}^{(0)}\|$, $c_1 = 0$ using gradient descent with a sufficiently small step size. Include the initialization and its norm and the final weights. What will $\boldsymbol{\beta}$ go to?

For the following problems, you just have to run the code in the Jupyter Notebook, which is essentially a demo. Run it, understand the plots, and answer these questions:

(d) (In MAML for regression using closed-form solutions) Considering the plot of regression test loss versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are set to 1?** Additionally, **how does their performance compare to the oracle**, which performs regression using only the features present in the data? Can you **explain the reason for the downward spike observed at `n_train_post = 32`**?

(e) By examining the changes in feature weights over time during meta-learning, **can you justify the observed improvement in performance?** Specifically, can you **explain why certain feature weights are driven towards zero**?

(f) (In MAML for regression using gradient descent) **With `num_gd_steps = 5`, does meta-learning contribute to improved performance during test time? Furthermore, if we change `num_gd_steps` to 1, does meta-learning continue to function effectively?**

(g) (In MAML for classification) Based on the plot of classification error versus `n_train_post`, **how does the performance of the meta-learned feature weights compare to the case where all feature weights are 1? How does the performance of the meta-learned feature weights compare to the oracle** (which performs logistic regression using only the features present in the data)?

(h) By observing the evolution of the feature weights over time as we perform meta-learning, **can you justify the improvement in performance? Specifically, can you explain why some feature weights are being driven towards zero?**

5. Example Difficulty and Early Exit

Deep Learning Practioners have recognized that within the same task, particular examples in the test set can actually be harder to perform predictions on that others. Why is that? What kinds of things are easier to learn? We explore the notion of example difficulty, proposed by Baldock et. al. that will allow us to perform deeper investigations on the topic. Furthermore, utilizing concepts from example difficulty can aid in development of techniques than can boost inference speeds, sample efficiency, and other key properties that allow scalability in both the micro and macro levels.

(a) Please run the notebook cells of `example_difficulty_hw.ipynb`. Don't forget to download the data at `data.npy` and `test_data.npy` and place these files in the same directory as the notebook. Note that the notebook could take a while to run, so start the notebook and work on something else while it runs. Then answer the following conceptual questions.

  i. **Briefly explain what example difficulty is in your own words. What is the setup for us to analyze example difficulty?**

  ii. **What kinds of properties do you think will make an example from this dataset difficult?**

  iii. **Why do we need to train the ResNet to convergence to be able to analyze the example difficulty?**

  iv. **In pytorch, what does adding hooks do? Why do we need to use them?**

  v. **Why do we train KNN classifiers at each layer of the ResNet? What data are we training them on?**

  vi. **Why do you think so many examples exit in the earlier layers?**

  vii. **Explain why the accuracy of the ResNet has a negative relationship with the prediction layer.**

  viii. **Explain the distinction between the hard and easy examples. Does this surprise you?**

  ix. **What kinds of patterns do you notice? Based on the composition of the layers, does it make sense?**

(b) Please fill in the notebook cells in `early_exit_hw_blank.ipynb`. Don't forget to download the data at `data.npy` and `test_data.npy` and place these files in the same directory as the notebook. Also include `architectures.py` in the same directory. Then answer the following conceptual questions.

  i. **What was the accuracy with a regular ResNet? Inference Speed? Total MACS?**

  ii. **What was the accuracy with an early exit ResNet-18? Inference Speed? Total MACS?**

  iii. **How did early exit do? Compare accuracy and MACS.**

  iv. **What was the lowest MACs you found. What might this say?**

  v. **When would we use early exit, versus just using a smaller model? What factors should we consider?**

6. TinyML - Quantization and Pruning.

(This question has been adapted with permission from MIT 6.S965 Fall 2022)

TinyML aims at addressing the need for efficient, low-latency, and localized machine learning solutions in the age of IoT and edge computing. It enables real-time decision-making and analytics on the device itself, ensuring faster response times, lower energy consumption, and improved data privacy.

To achieve these efficiency gains, techniques like quantization and pruning become critical. Quantization reduces the size of the model and the memory footprint by representing weights and activations with fewer bits, while pruning eliminates unimportant weights or neurons, further compressing the model.

(a) Please complete `pruning.ipynb`. Please upload this notebook to colab before running it. Then answer the following questions.

i. In part 1 the histogram of weights is plotted. **What are the common characteristics of the weight distribution in the different layers?**

ii. **How do these characteristics help pruning?**

iii. After viewing the sensitivity curves, please answer the following questions. **What's the relationship between pruning sparsity and model accuracy? (i.e., does accuracy increase or decrease when sparsity becomes higher?)**

iv. **Do all the layers have the same sensitivity?**

v. **Which layer is the most sensitive to the pruning sparsity?**

vi. (Optional) After completing part 7 in the notebook, please answer the following questions. **Explain why removing 30 percent of channels roughly leads to 50 percent computation reduction.**

vii. (Optional) **Explain why the latency reduction ratio is slightly smaller than computation reduction.**

viii. (Optional) **What are the advantages and disadvantages of fine-grained pruning and channel pruning? You can discuss from the perspective of compression ratio, accuracy, latency, hardware support (*i.e.*, requiring specialized hardware accelerator), etc.**

ix. (Optional) **If you want to make your model run faster on a smartphone, which pruning method will you use? Why?**

(b) Please complete `quantization.ipynb`. Please upload this notebook to colab before running it. Then answer the following questions.

i. After completing K-means Quantization, please answer the following questions. **If 4-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

ii. **If n-bit k-means quantization is performed, how many unique colors will be rendered in the quantized tensor?**

iii. After quantization aware training we see that even models that use 4 bit, or even 2 bit precision can still perform well. **Why do you think low precision quantization works at all?**

iv. (Optional) Please read through and complete up to question 4 in the notebook, then answer this question.

Recall that linear quantization can be represented as $r = S(q - Z)$. Linear quantization projects the floating point range $[fp_{min}, fp_{max}]$ to the quantized range $[quantized_{min}, quantized_{max}]$.

That is to say,

$$r_{\text{max}} = S(q_{\text{max}} - Z)$$
$$r_{\text{min}} = S(q_{\text{min}} - Z)$$

Substracting these two equations, we have,

$$S = r_{\text{max}}/q_{\text{max}}$$

$$S = (r_{\max} + r_{\min})/(q_{\max} + q_{\min})$$
$$S = (r_{\max} - r_{\min})/(q_{\max} - q_{\min})$$
$$S = r_{\max}/q_{\max} - r_{\min}/q_{\min}$$

Which of these is the correct result of subtracting the two equations?

v. (Optional) Once we determine the scaling factor S, we can directly use the relationship between $r_{\min}$ and $q_{\min}$ to calculate the zero point Z.

$$Z = \mathrm{int}(\mathrm{round}(r_{\min}/S - q_{\min})$$
$$Z = \mathrm{int}(\mathrm{round}(q_{\min} - r_{\min}/S))$$
$$Z = q_{\min} - r_{\min}/S$$
$$Z = r_{\min}/S - q_{\min}$$

Which of these are the correct zero point?

vi. (Optional) After finishing question 9 on the notebook, **please explain why there is no ReLU layer in the linear quantized model.**

vii. (Optional) After completing the notebook, **please compare the advantages and disadvantages of k-means-based quantization and linear quantization.** You can discuss from the perspective of accuracy, latency, hardware support, etc.

7. Homework Process and Study Group

Citing sources and collaborators are an important part of life, including being a student!
We also want to understand what resources you find helpful and how much time homework is taking, so we can change things in the future if possible.

(a) **What sources (if any) did you use as you worked through the homework?**

(b) **If you worked with someone on this homework, who did you work with?**
List names and student ID's. (In case of homework party, you can also just describe the group.)

(c) **Roughly how many total hours did you work on this homework?**

Contributors:

- Suhong Moon.
- Yuxi Liu.
- Bryan Wu.
- Olivia Watkins.
- Romil Bhardwaj.
- Anant Sahai.
- Saagar Sanghavi.
- Vignesh Subramanian.
- Josh Sanz.
- Ana Tudor.

- Mandi Zhao.
- Liam Tan.
- Yujun Lin.
- Ji Lin.
- Zhijian Liu.
- Song Han.