EECS 182 Deep Neural Networks
Spring 2023 Anant Sahai

Homework 10

This homework is due on Friday, April 21, 2023, at 10:59PM.

1. Meta-learning for Learning 1D functions

A common toy example with Neural Networks is to learn a 1D function. Suppose now that our task is not to learn not just one 1D function, but any of a class of 1D functions drawn from a *task distribution* $\mathcal{D}_T$.

In this problem we consider all signals of the form

$$y = \sum_{s \in \mathcal{S}} \alpha_s \phi_s^u(x)$$

The task distribution produces individual tasks which have true features with random coefficients in some *a priori* unknown set of indices $\mathcal{S}$. We do not yet know the contents of $\mathcal{S}$, but we can sample tasks from $\mathcal{D}_T$.

The important question is thus, **how do we use sampled tasks in training to improve our performance on an unseen task drawn from $\mathcal{D}_T$ at test time?**

One solution is to use our training tasks to learn a set of weights to apply to the features before performing regression through meta-learning. That is, we choose feature weights c_k to apply to the features $\phi_k^u(x)$ before learning coefficients $\hat{\beta}_k$ such that

$$\hat{y} = \sum_{k=0}^{d-1} \hat{\beta}_k c_k \phi_k^u(x).$$

These feature weights c_k are a toy model for the deep network that precedes the task-specific final layer in meta-learning.

We can then perform the min-norm optimization

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

$$\text{s.t. } \mathbf{y} = \sum_{k=0}^{d-1} \beta_k c_k \boldsymbol{\Phi}_k^u \tag{2}$$

where $\boldsymbol{\Phi}^u$ is the column vector of features $[\phi_0^u(x), \phi_1^u(x), \ldots, \phi_{d-1}^u(x)]^\top$ which are orthonormal with respect to the test distribution.

Now, we want to **learn c** which minimizes the expectation for $\hat{\boldsymbol{\beta}}$ over all tasks,

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

where $\mathcal{L}_T \left(\hat{\boldsymbol{\beta}}_T, \mathbf{c} \right)$ is the loss from learning $\hat{\boldsymbol{\beta}}$ for a specific task with the original formulation and a given $\mathbf{c}$ vector. $\mathbf{c}$ is shared across all tasks and is what we will optimize with meta-learning.

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)[1] 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. 1.

```
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 1: 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. 2 illustrates the path the weights take with these updates.

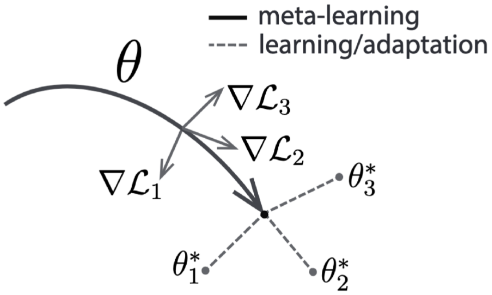

Figure 2: MAML gradient trajectory illustration

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.

(a) In the original MAML algorithm, the inner loop performs gradient descent to optimize loss with respect to a task distribution. However, here we're going to use the closed form min-norm solution for regression instead of gradient descent.

[1] 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*

Let's recall the closed form solution to the min-norm problem. Write the solution to

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

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

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

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

Show that in this case, the solution to the min-norm problem 1 is $\hat{\boldsymbol{\beta}} = \frac{1}{c_0^2+c_1^2}\begin{bmatrix} c_0 \\ c_1 \end{bmatrix}$

(c) 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:**

$$\frac{\mathrm{d}}{\mathrm{d}\mathbf{c}}\left(\mathbb{E}_{x_{test}, y_{test}}\left[\frac{1}{2}\left\|y - \hat{\beta}_0 c_0 \phi_t^u(x) - \hat{\beta}_1 c_1 \phi_a^u(x)\right\|_2^2\right]\right).$$

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

(d) **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?

Run the code in the Jupyter Notebook and then answer these questions:

(e) (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`**?

(f) 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**?

(g) (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?**

(h) (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)?

(i) 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?**

2. Vision Transformer

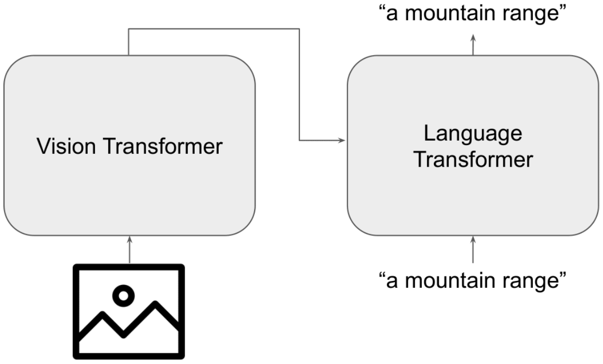

Figure 3: Image captioning model

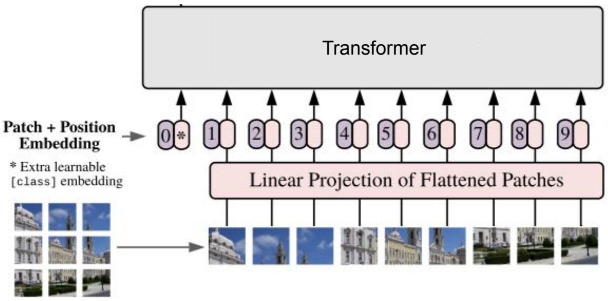
Transformer
Patch + Position Embedding
* Extra learnable [class] embedding
0 *
1
2
3
4
5
6
7
8
9
Linear Projection of Flattened Patches

Figure 4: Vision Transformer

You are building a model to perform image captioning. As shown in Figure 3, the model consists of a vision transformer which takes in images and a language transformer which outputs captions. The language transformer will use cross-attention to access the representation of the image.

(a) For each transformer, state whether it is more appropriate to use a transformer encoder (a transformer with no masking except to handle padding) or decoder (a transformer with autoregressive self-attention masking) and why.

Vision transformer?

○ Encoder-style transformer

○ Decoder-style transformer

Reason:

Language transformer?

○ Encoder-style transformer

○ Decoder-style transformer

Reason:

(b) A standard language transformer for captioning problems alternates between layers with cross-attention between visual and language features and layers with self-attention among language features. Let's say we modify the language transformer to have a single layer which performs both attention operations at once. The grid below shows the attention mask for this operation. (For now, assume the vision transformer only outputs 3 image tokens called <ENC1>, <ENC2>, and <ENC3>. <SOS> is the start token, and <PAD> is a padding token.)

(i) One axis on this grid represents sequence embeddings used to make the queries, and the other axis represents sequence embeddings used to make the keys. **Which is which?**

○ Each column creates a query, each row creates a key and a value
○ Each column creates a key and a value, each row creates a query
○ Each column creates a query and a value, each row creates a key
○ Each column creates a key, each row creates a query and a value

(ii) **Mark X in some of the blank cells in the grid to illustrate the attention masks.** (A X marked cell is masked out, a blank cell is not.)

	<SOS>	a	mountain	range	<PAD>
<SOS>					
a					
mountain					
range					
<PAD>					
<ENC1>					
<ENC2>					
<ENC3>					

(c) In discussion, we showed that the runtime complexity of vision transformer attention is $O(D(H^4/P^4))$, where H is the image height and width, P is the patch size, and D is the feature dimension of the queries, keys, and values. Some recent papers have reduced the complexity of vision transformer attention by segmenting an image into windows, as shown in Figure 5.

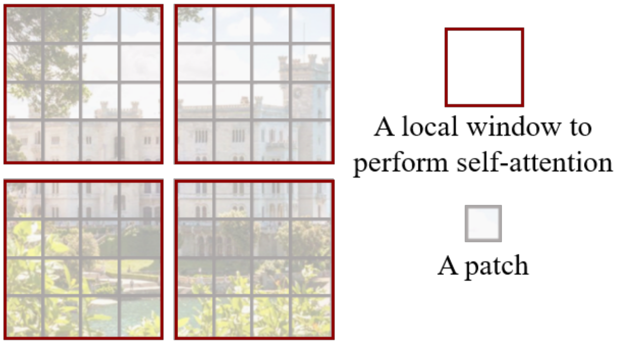

Figure 5: Vision transformer attention with windows

Patches only attend to other patches within the same window. **What is the Big-O runtime complexity of the attention operation after this modification?** Assume each window consists of K by K patches.

3. Pretraining and Finetuning

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 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. If we could only afford to fine-tune one layer, **which layer (i.e. 1,2,3,4,5) in Figure 6 should we choose to finetune to get the best performance on CIFAR-10-C? Give brief intuition as to why.**

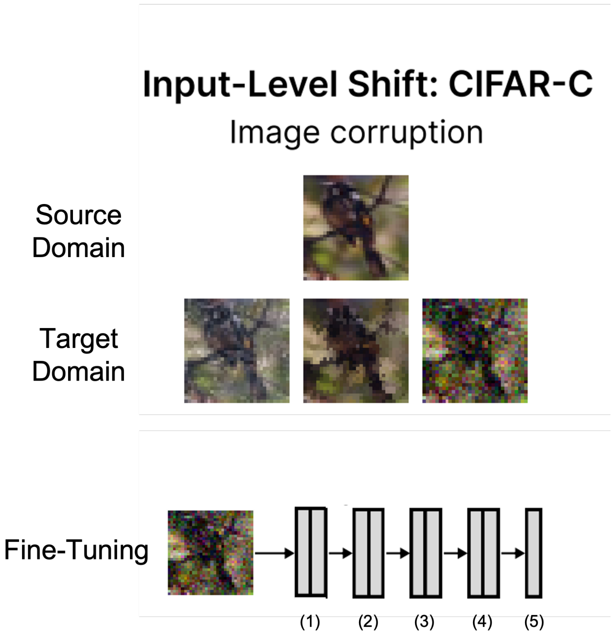

Figure 6: Fine-tuning the model pretrained with CIFAR-10 on CIFAR-10-C dataset

4. 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?**

5. 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.)

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, 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, 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,

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

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? Write it down here where you'll need to remember it for the self-grade form.**

Contributors:

- Anant Sahai.
- Saagar Sanghavi.
- Vignesh Subramanian.
- Josh Sanz.
- Ana Tudor.
- Mandi Zhao.
- Olivia Watkins.
- Suhong Moon.
- Bryan Wu.
- Romil Bhardwaj.
- Yujun Lin.

- Ji Lin.
- Zhijian Liu.
- Song Han.
- Liam Tan.