

This homework is due on December 1, at 10:59PM.

1. Debugging Transformers

You're implementing a Transformer encoder-decoder model for document summarization (a sequence-to-sequence NLP task). You write the initialization of your embedding layer and head weights as below:

```

1 class Transformer(nn.Module):
2     def __init__(self, n_words, max_len, n_layers,
3                 d_model, n_heads, d_ffn, p_drop):
4         super().__init__()
5         self.emb_word = nn.Embedding(n_words, d_model)
6         self.emb_pos = nn.Embedding(max_len, d_model)
7
8         # Initialize embedding layers
9         self.emb_word.weight.data.normal_(mean=0, std=1)
10        self.emb_pos.weight.data.normal_(mean=0, std=1)
11
12        self.emb_ln = nn.LayerNorm(d_model)
13        self.encoder_layers = nn.ModuleList([
14            TransformerLayer(False, d_model, n_heads, d_ffn, p_drop)
15            for _ in range(n_layers)
16        ])
17        self.decoder_layers = nn.ModuleList([
18            TransformerLayer(True, d_model, n_heads, d_ffn, p_drop)
19            for _ in range(n_layers)
20        ])
21        self.lm_head = nn.Linear(d_model, n_words)
22        # Share lm_head weights with emb_word
23        self.lm_head.weight = self.emb_word.weight
24        self.criterion = nn.CrossEntropyLoss(ignore_index=-100)

```

After training this model, you compare your implementation with your friend's by looking at the loss curves:

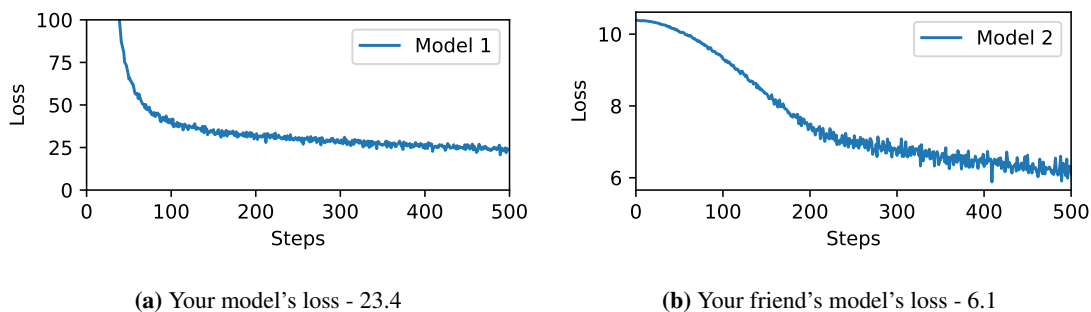


Figure 1: Comparing your model's loss vs your friend's model's loss. Your model is doing significantly worse.

Your friend suggests that there's something wrong with how the head gets initialized. **Identify the bug in your initialization, fix it by correcting the buggy lines, and briefly explain why your fix should work.**

Hint: remember that `d_model` is large in transformer models.

Hint: your change needs to impact line 9 somehow since that is where the head is initialized.

The bug: (brief description)

The fix (code): (Just show anything you change and/or add to the code.)

Why the fix should work: (brief explanation)

2. Comparing Distributions

Divergence metrics provide a principled measure of difference between a pair of distributions (P, Q) . One such example is the Kullback-Leibler Divergence, that is defined as

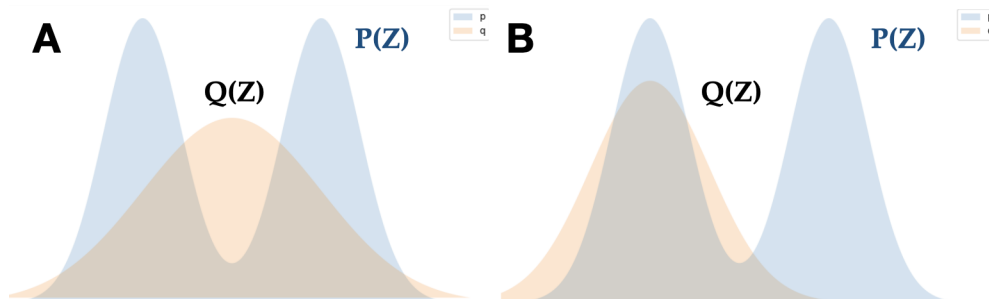
$$D_{\text{KL}}(P||Q) = \mathbb{E}_{x \sim P(x)} \left[\log \frac{P(x)}{Q(x)} \right]$$

- (a) Technically D_{KL} is not a true distance since it is asymmetric, i.e. generally $D_{\text{KL}}(P||Q) \neq D_{\text{KL}}(Q||P)$.

Give an example of univariate distributions P and Q where $D_{\text{KL}}(P||Q) \neq \infty$, $D_{\text{KL}}(Q||P) = \infty$.

- (b) For a fixed target distribution P , we call $D_{\text{KL}}(P||Q)$ the *forward-KL*, while calling $D_{\text{KL}}(Q||P)$ the *reverse-KL*. Due to the asymmetric nature of KL, distributions Q that minimize $D_{\text{KL}}(P||Q)$ can be different from those minimizing $D_{\text{KL}}(Q||P)$.

From the following plots, **identify which of (A, B) correspond to minimizing forward vs. reverse KL. Give brief reasoning.** Here, only the mean and standard deviation of Q is allowed to vary during the minimization.



3. Variational Information Bottleneck

In class, you saw tricks that we introduced in the context of Variational Auto-Encoders to allow ourselves to get the latent space to have a desirable distribution. It turns out that we can use the same spirit even with tasks different than auto-encoding.

Consider a prediction task that maps an input source X to a target Y through a latent variable Z , as shown in the figure below. Our goal is to learn a latent encoding that is maximally useful for our target task, while trying to be close to a target distribution $r(Z)$.

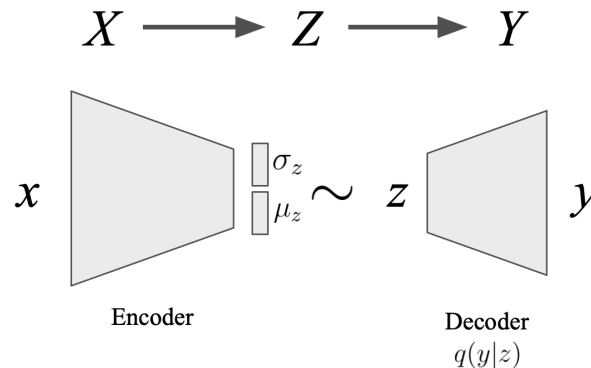


Figure 2: Overview of a VIB that maps an input X to the target Y through a latent variable Z (top). We use deep neural networks for both the encoder and task-relevant “decoder.”

- (a) Assume that we decide to have the encoder network (parameterized by θ_e) take both an input x and some independent randomness V to emit a random sample Z in the latent space drawn according to the Gaussian distribution $p_{\theta_e}(Z|x)$.

For this part, assume that we want Z to be a scalar Gaussian (conditioned on x) with mean μ and variance σ^2 with the encoder neural network emitting the two scalars μ and σ as functions of x . Assume that V is drawn from iid standard $\mathcal{N}(0, 1)$ Gaussian random variables.

Draw a block diagram with multipliers and adders showing how we get Z from μ and σ along with V .

- (b) Assume that our task is a classification-type task and the “decoder” network (parameterized by θ_d) emits scores for the different classes that we run through a softmax to get the distribution $q_{\theta_d}(y|z)$ over classes when the latent variable takes value z .

To train our networks using our N training points, we want to use SGD to approximately minimize the following loss:

$$L = \frac{1}{N} \sum_{n=1}^N \mathbb{E}_{z \sim p_{\theta_e}(z|x_n)} \left[\underbrace{-\log q_{\theta_d}(y_n|z)}_{\text{task loss}} \right] + \beta \overbrace{D_{KL}(p_{\theta_e}(Z|x_n) || r(Z))}^{\text{latent regularizer}} \quad (1)$$

where the y_n is the training label for input x_n and during training, we draw fresh randomness V each time we see an input, and we set $r(Z)$ to be a standard Gaussian $\mathcal{N}(0, 1)$.

If we train using SGD treating the random samples of V as a part of the external input, **select all the loss terms that contribute (via backprop) to the gradients used to learn the encoder and decoder parameters:**

For encoder parameters θ_e : ☐ task loss ☐ latent regularizer

For decoder parameters θ_d : ☐ task loss ☐ latent regularizer

- (c) Let's say we implemented the above information bottleneck for the task of MNIST classification. **Which of the curves in Figure 3 below best represents the trend of the *validation error* (on held-out data) with increasing regularization strength parameter β ?** (select one)

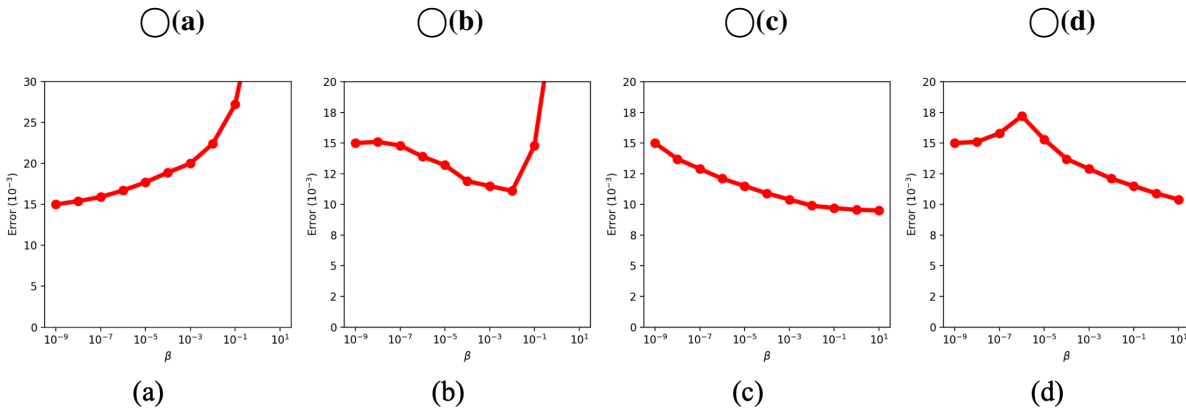


Figure 3: Validation error (on held-out data) profiles for different values of β .

- (d) Let's say we implemented the above information bottleneck for the task of MNIST classification for three digits, and set the dimension of the latent space Z to 2. Figure 4 below shows the latent space embeddings of the input data, with different symbols corresponding to different class labels, for three choices of $\beta \in \{10^{-6}, 10^{-3}, 10^0\}$. Now answer these two questions:

- i. **Guess the respective values of β used to generate the samples.** (select one for each fig)

(HINT: Don't forget to look at the axis labels to see the scale.)

(a) ☐ $\beta = 10^{-6}$ ☐ $\beta = 10^{-3}$ ☐ $\beta = 10^0$

(b) ☐ $\beta = 10^{-6}$ ☐ $\beta = 10^{-3}$ ☐ $\beta = 10^0$

(c) ☐ $\beta = 10^{-6}$ ☐ $\beta = 10^{-3}$ ☐ $\beta = 10^0$

- ii. **Which of the three experiments in Figure 4 results in a *better* latent space for the prediction task?** (select one)

☐ (a)

☐ (b)

☐ (c)

4. Variational Autoencoders

(Parts of this problem are adapted from *Deep Generative Models*, Stanford University)

For this problem we will be using PyTorch to implement the variational autoencoder (VAE) and learn a probabilistic model of the MNIST dataset of handwritten digits. Formally, we observe a sequence of binary pixels $\mathbf{x} \in \{0, 1\}^d$ and let $\mathbf{z} \in \mathbb{R}^k$ denote a set of latent variables. Our goal is to learn a latent variable model $p_\theta(\mathbf{x})$ of the high-dimensional data distribution $p_{data}(\mathbf{x})$.

The VAE is a latent variable model with a specific parameterization $p_\theta(\mathbf{x}) = \int p_\theta(\mathbf{x}, \mathbf{z}) d\mathbf{z} = \int p(\mathbf{z}) p_\theta(\mathbf{x}|\mathbf{z}) d\mathbf{z}$. Specifically, VAE is defined by the following generative process (often called **reparameterization trick**):

$$p(\mathbf{z}) = \mathcal{N}(\mathbf{z}|0, I)$$

(sample noise from standard Gaussian)

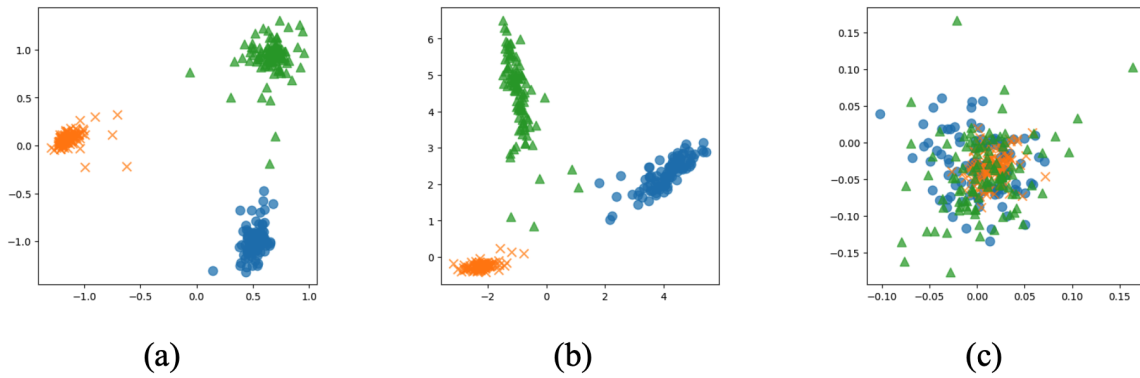


Figure 4: MNIST VIB with 2D latent space.

$$p_{\theta}(\mathbf{x}|\mathbf{z}) = \text{Bern}(\mathbf{x}|f_{\theta}(\mathbf{z})) \quad (\text{decode noise to generate sample from real-distribution})$$

That is, we assume that the latent variables \mathbf{z} are sampled from a unit Gaussian distribution $\mathcal{N}(\mathbf{z}|0, I)$. The latent \mathbf{z} are then passed through a neural network decoder $f_{\theta}(\cdot)$ to obtain the parameters of the d Bernoulli random variables that model the pixels in each image.

To learn the parameterized distribution we would like to maximize the marginal likelihood $p_{\theta}(\mathbf{x})$. However computing $p_{\theta}(\mathbf{x}) = \int p(\mathbf{z})p_{\theta}(\mathbf{x}|\mathbf{z})d\mathbf{z}$ is generally intractable since this requires integrating over all possible values of $\mathbf{z} \in \mathbb{R}$. Instead, we consider a variational approximation to the true posterior

$$q_{\phi}(\mathbf{z}|\mathbf{x}) = \mathcal{N}(\mathbf{z}|\mu_{\phi}(\mathbf{x}), \text{diag}(\sigma_{\phi}^2(\mathbf{x})))$$

In particular, we pass each image \mathbf{x} through a neural network that outputs mean μ_{ϕ} and diagonal covariance $\text{diag}(\sigma_{\phi}^2(\mathbf{x}))$ of the multivariate Gaussian distribution that approximates the distribution over the latent variables \mathbf{z} given \mathbf{x} . The high level intuition for training parameters (θ, ϕ) requires considering two expressions:

- **Decoding Latents** : Sample latents from $q_{\phi}(\mathbf{z})$, maximize likelihood of generating samples $\mathbf{x} \sim p_{data}$
- **Matching Prior** : A Kullback-Leibler (KL) term to constraint $q_{\phi}(\mathbf{z})$ to be close to the $p(\mathbf{z})$

Putting these terms together, gives us a lower-bound of the true marginal log-likelihood, called the **evidence lower bound (ELBO)**:

$$\log p_{\theta}(\mathbf{x}) \geq \text{ELBO}(\mathbf{x}; \theta, \phi) = \underbrace{\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})]}_{\text{Decoding Latents}} - \underbrace{D_{\text{KL}}(q_{\phi}(\mathbf{z}|\mathbf{x})||p(\mathbf{z}))}_{\text{Matching Prior}}$$

Open the [Jupyter Notebook](#). The notebook will depend on this [repository](#); you will implement the reparameterization trick in `q_vae_gan.ipynb` at the function `sample_gaussian`. Specifically, your answer will take in the mean m and variance v of the Gaussian and return a sample $\mathbf{x} \sim \mathcal{N}(m, \text{diag}(v))$

Then, implement `negative_elbo_bound` loss function.

Note: We ask for the negative ELBO, as PyTorch optimizers minimize the loss function. Further, since we are computing the negative ELBO over a mini-batch of data $\{x^{(i)}\}_{i=1}^n$, make sure to compute the average of per-sample ELBO. Finally, note that the ELBO itself cannot be computed exactly since computation of the reconstruction term is intractable. Instead, you should estimate the reconstruction term via Monte-Carlo sampling

$$-\mathbb{E}_{q_{\phi}(\mathbf{z}|\mathbf{x})}[\log p_{\theta}(\mathbf{x}|\mathbf{z})] \approx -\log p_{\theta}(\mathbf{x}|\mathbf{z}^{(1)})$$

where $\mathbf{z}^{(1)} \sim q_\phi(\mathbf{z}|\mathbf{x})$ denotes a single sample from the learned posterior.

The `negative_elbo_bound` expects as output three quantities: *average negative ELBO, reconstruction loss, KL divergence*.

Answer these questions:

- (a) Test your implementation by training a VAE using the notebook or the following command:

```
python experiment.py --model vae
```

Once the run is complete (10000 iterations), **report the following numbers** : the *average*

- negative ELBO
- KL-Divergence term
- reconstruction loss

Since we're using stochastic optimization, you may wish to run the model multiple times and **report each metric's mean and corresponding standard error** (*Hint: the negative ELBO on the test subset should be ~ 100*)

- (b) **Visualize 200 digits (generate a single image tiled in a grid of 10×20 digits) sampled from $p_\theta(\mathbf{x})$**

5. 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 $(x_1, y_1), (x_2, y_2), \dots, (x_H, y_H)$, making updates $\beta_0 \mapsto \beta_1 \mapsto \beta_2 \mapsto \dots \mapsto \beta_H$. 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: $\mathcal{L}_T(\hat{\beta}, c)$. Now the learner is destroyed, and the genome c is updated by backpropagating across the learner's entire life, right back to its birth.¹

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:

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

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:

¹Of 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...

- 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.
- $\mathcal{L}_T(\hat{\beta}, c)$: **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)² 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. 5.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks

Require: α, β : 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 5: 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 $\{\mathcal{T}_i\}$ 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. 6 illustrates the path the weights take with these updates.

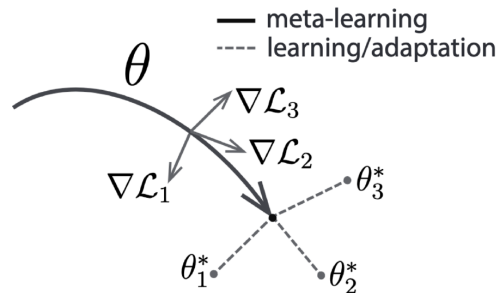


Figure 6: MAML gradient trajectory illustration

²C. 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*

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 $\mathbb{R} \rightarrow \mathbb{R}$. The space of all possible $\mathbb{R} \rightarrow \mathbb{R}$ functions is too vast to consider, so we only take a finite-dimensional subspace of it. Specifically, we pick d functions $\phi_0, \phi_1, \dots, \phi_{d-1}$, 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 $(x_1, f_\alpha(x_1)), \dots, (x_H, f_\alpha(x_H))$. Here, the numbers x_1, \dots, 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 $(x_1, f_\alpha(x_1)), \dots, (x_H, f_\alpha(x_H))$.

The learner performs minimal-norm regression on them to obtain $\hat{\beta}$ in one step³. 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:

$$\hat{\beta} = \operatorname{argmin}_{\beta} \|\beta\|_2^2 \tag{2}$$

$$\text{s.t. } y_i = \sum_{k=0}^{d-1} \beta_k c_k \phi_k(x_i) \quad \forall i = 1, 2, \dots, K. \tag{3}$$

Although 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 $\phi_0, \dots, \phi_{d-1}$, 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

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

³Least-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.

in terms of A and \mathbf{y} is

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

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:

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

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, \dots, 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 $\hat{\beta}$ using the training data. Note, both the coefficients and the feature weights are 2-d vectors.

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

- (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:**

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

Use the values for β 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 $i \rightarrow \infty$ 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 β 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?**

6. 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?**

The following are old exam questions simply for your reference. You do not have to do these questions. Their coverage is redundant.

7. Fine-tuning Large Models for Multiple Tasks

In the context of fine-tuning large pre-trained foundation models for multiple tasks, consider the following three scenarios:

- (1) Using a foundation model with different soft prompts tuned per task, while keeping the main model frozen. Assume prompts have a token length of 5.
- (2) Using a foundation model held frozen, with task-specific low-rank adapters (i.e. we train A, B matrices to allow us to replace the relevant weight matrix W with $W + AB$ where A is initialized to zero and B is randomly initialized) fine-tuned for the attention-parameters (key, value, and query weight matrices) in the top four layers.
- (3) Full-fine tuning of the entire model using the data from the multiple target tasks simultaneously.

You can assume that the foundation model has 13B parameters with a max context length of 512, hidden_dim 5120, Multi-head-attention with 40 heads and 40 layers, trained with a dataset consisting of 1T tokens.

- (a) **Which of these scenarios is most likely to lead to catastrophic forgetting?** (select one)
 - ☐ Scenario 1
 - ☐ Scenario 2
 - ☐ Scenario 3
 - ☐ None of the above
- (b) **What is the total number of trainable parameters using soft prompt tuning?**
- (c) Suppose we decide to use meta-learning to get better initializations for the A and B matrices to be used for task-specific low-rank adaptation. Assume you have a large family of tasks with lots of relevant training data. **Which meta-learning approach do you think will be more practically effective given this setting:** (select one)
 - ☐ MAML with a number of inner iterations k of around 10. (Recall that MAML requires you to compute the gradients to the initial condition before the inner training iterations through the training iterations but on held-out test data.)
 - ☐ REPTILE using a large batch of task-specific data at a time. (Recall that REPTILE uses the net movement from the initial condition of this meta-iteration as an approximation for the meta-gradient and just moves the meta-learned initial-condition a small step in that direction.)
- (d) To better defend against catastrophic forgetting during your meta-learning approach in the previous part, **which of the following would you consider doing:** (mark all that apply)
 - ☐ During meta-learning, include occasional gradient updates to A and B on random subsets of the original training data with the original self-supervised training loss.
 - ☐ During meta-learning, include occasional gradient updates to A and B using the original self-supervised training loss but on the new training data from the training family of tasks.
 - ☐ After your meta-learning updates are done, reinitialize the A matrices to zero before actually fine-tuning on new tasks.

Contributors:

- Linyuan Gong.
- Romil Bhardwaj.
- Kumar Krishna Agrawal.
- Dhruv Shah.
- Anant Sahai.
- Aditya Grover.
- Stefano Ermon.
- Saagar Sanghavi.
- Vignesh Subramanian.
- Josh Sanz.
- Ana Tudor.
- Mandi Zhao.
- Yuxi Liu.