Deja Vu: Continual Model Generalization over Unseen Domains

Pranshu Agarwal (230776)
Chaitanya Vishwas Brahmapurikar (230305)
Sunij Singh Gangwar (231051)
Om Ji Gupta (230719)

Indian Institute of Technology Kanpur

November 26, 2024

Course: CS771
Professor: Prof. Piyush Rai.

Introduction

In real world, there are situations when the deep learning models run into non-stationary environments where the distribution of the data changes with time. We'd like to do do better on problems like:

- Surveillance cameras used for environmental monitoring can work normally with excellent performance on clear days, but have inferior performance or even become "blind" when the weather turns bad or the lighting conditions become poor as they cannot "adapt" and "generalize" to a changing environment. (Bak et al., 2018)
- Another potent example could be as follows: Consider conducting lung imaging analysis for corona-viruses deep learning models may present do excellent after being trained on a large number of samples for certain variant (e.g., the Alpha variant of COVID- 19), but are difficult to provide accurate and timely analysis for later variants (e.g., the Delta or Omicron variant) when they just appear. (Singh et al., 2020)

Some Terminologies

This paper discusses an important problem in practical scenarios with continual domain shifts, i.e., to improve model performance before and during training on a new target domain, in what we call the **Unfamiliar Period**. We also aim to achieve good model performance after training, providing the model with capabilities of target domain adaptation and forgetting alleviation.

- **Target Domain Generalization (TDG):** Model generalization performance on a new target domain *before and during* its training.
- **Target Domain Adaptation (TDA):** Model performance on a trained domain *right* after its training.
- **Forgetting Alleviation (FA):** Model performance on a trained domain *after* the model is trained with other domains.

Problem Formulation

We assume that the label source domain is:

$$\mathcal{S} = \{(x_i, y_i) \mid x_i \sim \mathcal{P}_X^S, y_i \sim \mathcal{P}_Y^S\}_{i=1}^{N_S}$$

Here $\mathcal{P}_X$ and P_Y are the input and label distributions, while $\mathcal{S}$ is the sampling quantity. Also consider a sequence of continually arriving target domains $\boldsymbol{T} = \{\mathcal{T}^t\}_{t=1}^{T}$ where T denotes the total number of domains. The t-th domain contains $N_{\mathcal{T}^t}$ data samples **x** but no labels **y**, i.e.,

$$\mathcal{T}^t = \{x_i \mid x_i \sim \mathcal{P}_X^{\mathcal{T}^t}\}_{i=1}^{N_{\mathcal{T}^t}}$$

We assume that the model is a deep neural network, and without loss of generality, the neural network consists of a feature extractor f_θ at the bottom and a classifier g_ω at the top. We shall assume that for each dataset $\mathcal{T}_t$, in addition to $f_\theta^t \circ g_\omega^t$, the previous version $f_\theta^{t-1} \circ g_\omega^{t-1}$ is also known.

RaTP Framework Overview

We first discuss some of the important modules we will use:

- **Random Mixture Augumentation**: Training-free data augmentation for robust generalization.
- **T2PL (Top2 Pseudo Labeling)**: Generates high-confidence pseudo-labels for the target domain.
- **Prototype Contrastive Alignment (PCA)**: Aligns domain prototypes using contrastive learning.

Let us see each of these in greater detail.

Random Mixture Augmentation

The basic idea is to create "noisy" training examples *around* a given training example for the purpose of regularization. We shall use N_{aug} number of simple auto-encoders $\boldsymbol{R} = \{R_i\}_{i=1}^{N_{\text{aug}}}$ where each encoder consists of an encoder e_ζ and d_η for generation of synthetic training examples and labels. Note that since auto-encoders work on the principle of minimizing the quantity $\|\mathbf{x} - \hat{\mathbf{x}}\|$ where $\hat{\mathbf{x}} = d_\eta(e_\zeta(\mathbf{x}))$, hence addition of the noisy $\hat{\mathbf{x}}$ to the training set makes it more robust! To add even more randomness, consider the following algorithm: AdaIN to inject noise to auto-encoder. This contains two linear layers l_{ϕ_1}, l_{ϕ_2} (basically combination of two linear models, nothing fancy), and when a noisy input n is drawn from the normalized distribution Uniform(0,1), they produce two corresponding noisy outputs (with smaller variances).

Random Mixture Augmentation

These two are injected into the representation as multiplicative and noisy outputs as follows:

$$R(\mathbf{x}) = d_\eta(l_{\phi_1}(n) \cdot l_{\text{IN}}(e_\zeta(x)) + l_{\phi_2}(n))$$

where $l_{\text{IN}}(\cdot)$ represents element-wise normalization layer. **RandMix** works as feeding training data to all encoders and mixing outputs with random weights from the normalized distribution as

$$\mathbf{w} = \{w_i \mid w_i \sim \text{Uniform}(0,1)_{i=0}^{N_{\text{aug}}}$$

. The weighted "noise" is added to the original input **x** and then scaled by the sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$ to yield the following:

$$\mathbf{R}(x) = \sigma\left(\frac{1}{\sum_{i=0}^{N_{\text{aug}}} w_i}\left[w_0 x + \sum_{i=1}^{N_{\text{aug}}} w_i R_i(x)\right]\right).$$

Random Mixture Augmentation

Now for each subsequent domain, the auto-encoder parameters (ζ, η), AdaIN-s (ϕ_1, ϕ_2, n) and mix-up weights **w** are initialized all over again. We generate the augmentation data ("noisy" data) corresponding to a "mini-batch" of the inputs from the current domain and fed with original inputs into the model for training. Although note that for the source domain, we use all the training inputs for the augmentation. While we can use full training inputs in unlabeled domains with approaches like "pseudo-labels" the supervision is unreliable and accurate. Therefore, to select which mini-batch to use for augmentation, we use something called "prediction confidence" as follows:

$$\hat{\mathbf{x}} = \begin{cases} \mathbf{R}(\mathbf{x}) & \text{if } \max[g_\omega(f_\theta(\mathbf{x}))]_K \geq r_{\text{con}} \\ \emptyset & \text{otherwise} \end{cases}$$

Random Mixture Augmentation

Recall that the classifier g_ω can be thought of as a vector of probabilities of belonging to different classes. If it crosses a threshold of r_{con} we are "confident" that it belongs to that particular class (hence the name, confidence threshold!). Therefore, we simply augment inputs with higher prediction confidence. Let us discuss another important algorithm involved the framework of **RaTP**.

Top Pseudo Labelling

We know that all the target domains arrive with only the data but no label. Then how do we reasonably provide accurate supervision for subsequent optimizations? The answer is **Pseudo labeling**! In each subsequent domains, after the prediction, we assign pseudo labels based on the prediction confidence of training inputs. Specifically, pick top 50% of the most confident predictions for each $k \in \{1, 2, 3, ...\}$ as follows:

$$\mathcal{I}_k = \cup^{\frac{N_{\mathcal{T}^t}}{r_{\text{top}} \cdot K}} \{\text{arg max}_{\mathbf{x} \in \mathcal{T}^t} [g_{\omega,k}(f_\theta(\mathbf{x}))]\}$$

If the data is class-balanced, the value of r_{top} would be 2. Here $g_{\omega,k}$ is k-th element of classifier outputs. Therefore all the inputs for which we are confident enough about their class would be:

$$\mathcal{F} = \cup_{k=1}^{K} \left\{ \cup_{i \in \mathcal{I}_K, \mathbf{x} \in \mathcal{T}^t} \{\mathbf{x}_i\} \right\}$$

Top Pseudo Labeling

Now let us find the centroid centroid for each class as the weighted average of prediction confidence:

$$p_k = \frac{\sum_{\mathrm{x}_i \in \mathcal{F}} g_{\omega,k}(f_\theta(\mathrm{x}_i)) \cdot f_\theta(\mathrm{x}_i)}{\sum_{\mathrm{x}_i \in \mathcal{F}} g_{\omega,k}(f_\theta(\mathrm{x}_i))}$$

. Note that our work is not done yet, if we classify all the inputs based on these class centroids, then this may give inaccurate results in case of "overlapped" data, wherein the close vicinity of class means there may lie inputs with different labels. We need to do more work: we compute similarity of all class means with the unlabeled data and select, say, top 50% of training inputs most similar to them. Now we apply kNN on these top 50% to take approximately 5% around them to classify the training inputs. Let us write this down formally as:

$$\mathcal{I}'_K = \cup^{\frac{N_{\mathcal{T}^t}}{r_{\text{top}} \cdot K}} \left\{ \arg\max_{\mathbf{x} \in \mathcal{T}^t} \left[\frac{f_\theta(\mathbf{x}) \cdot p_k^T}{\| f_\theta(\mathbf{x}) \cdot \| p_k \|} \right] \right\}$$

Top Pseudo Labelling

We select the top 50% most similar unlabeled training inputs to class mean for each k, hence the r_{top} would be 2. The set of such inputs would be

$$\mathcal{F}' = \cup_{k=1}^{K} \left\{ \cup_{i \in \mathcal{I}'_k, \mathbf{x} \in \mathcal{T}^t} \{(\mathbf{x}_i, k)\} \right\}$$

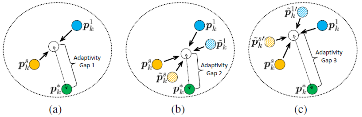
Now based on kNN the pseudo label would be

$$\hat{\mathbf{y}} = (\mathbf{x}, \mathcal{F}')_{\text{Euclidean}}^{\frac{N_{\mathcal{T}^t}}{r'_{\text{top}} \cdot K}}$$

The value of $\frac{N_{\mathcal{T}^t}}{r_{\text{top}} \cdot K}$ would be the number of closest neighbours kNN will look at (to account for top 5% neighbours, value of would be $r'_{\text{top}} = 20$, assuming similar class size, otherwise need to be tuned, or could choose different for different classes)

Our bread and butter: Prototype Contrastive Alignment

This algorithm aims to align domains together to improve our model generalization ability. We could use PL (Prototype Learning) naively as follows: first consider $\mathbf{p}^t = \{p_k^t\}_{k=1}^K$ for a certain domain $\mathcal{T}^t$, now to achieve cross-domain alignment, prototypes of different domains for the same classes are aggregated and averaged. The data samples from different domains are then optimized to approach these domain-average prototypes. However, this faces something called **adaptivity gap**.



$\boldsymbol{p}_k^1$
$\boldsymbol{p}_k^s$
$\boldsymbol{p}_k^*$
Adaptivity Gap 1
$\tilde{\boldsymbol{p}}_k^1$
$\tilde{\boldsymbol{p}}_k^s$
Adaptivity Gap 2
$\tilde{\boldsymbol{p}}_k^{1\prime}$
$\tilde{\boldsymbol{p}}_k^{s\prime}$
Adaptivity Gap 3
(a)
(b)
(c)

Figure 2: (a): When the model encounters a domain whose prototypes ($\boldsymbol{p}_k^1$) are far from the optimal ones ($\boldsymbol{p}_k^*$), domain alignment with regular prototype learning will enlarge the adaptivity gap. (b): `RandMix` may be helpful to reduce the gap in many cases of ($\tilde{\boldsymbol{p}}_k^s, \tilde{\boldsymbol{p}}_k^1$). (c): However, in some cases `RandMix` may produce low-quality data augmentation ($\tilde{\boldsymbol{p}}_k^{s\prime}, \tilde{\boldsymbol{p}}_k^{1\prime}$) that negatively affects the regular prototype learning and enlarges the gap. Here, we have Adaptivity Gap 3 > Adaptivity Gap 1 > Adaptivity Gap 2.

Figure: Adaptivity Gap

PCA

Now let us try a different approach. With the help of pseudo labeling we already have p_k^t of current domain as well as p_k^{t-1} of the previous domain. The main idea is that we want to “change” the feature representation of $\mathbf{x}$ so that $\mathbf{x}$ in new feature space will get even “closer” to our class means. Let us proceed to establish this mathematically:

$$\mathcal{L}_{\text{CE}} = \mathbb{E}_{\mathbf{x}_i \sim \mathcal{P}_X^t} \left[-\sum_{k=1}^{K} \mathbb{I}_{\hat{y}_i = k} \log \frac{\exp\left(p_k^{t\top} f_\theta(\mathbf{x}_i) + b_k\right)}{\sum_{c=1}^{K} \exp\left(p_c^{t\top} f_\theta(\mathbf{x}_i) + b_c\right)} \right]$$

Here we view neuron weights of the classifer ω as the prototypes and b_k and b_c are the biases which we can set to 0 during training. Also we would like to increase the increase the similarity of $f_\theta(\mathbf{x}_i)$ with p_k^t as well as p_k^{t-1} to enhance the distinguishability of different classes. We can formulate the prototype contrastive alignment as follows:

$$\mathcal{L}_{PCA} = \mathbb{E}_{\mathbf{x}_i \sim \mathcal{P}_X^t} \left\{ \sum_{k=1}^{K} -\mathbb{I}_{\hat{y}_i = k} \log \frac{\exp\left(p_k^{t\top} f_\theta(\mathbf{x}_i)\right) + \exp\left(p_k^{t-1\top} f_\theta(\mathbf{x}_i)\right)}{\Delta} \right\}$$

where Δ is noting but

$$\Delta = \sum_{c=1}^{K} \exp\left(p_c^{t\top} f_\theta(\mathbf{x}_i)\right) + \sum_{c=1}^{K} \exp\left(p_c^{t-1\top} f_\theta(\mathbf{x}_i)\right)$$

$$+ \sum_{\mathbf{x}_j \in T^t, j \neq i} \mathbb{I}_{\hat{\mathbf{y}}_i \neq \hat{\mathbf{y}}_j} \exp\left(f_\theta(\mathbf{x}_i)^\top f_\theta(\mathbf{x}_j)\right)$$

The aim of the above expression is to maximize the similarity of representation of $f_\theta(\mathbf{x}_i)$ with p_k^t and p_k^{t-1} among all the similarities. We also need to adopt knowledge distillation from stored old model to our current model for forgetting compensation. Hence, our the final objective would be to minimize:

$$\mathcal{L} = \mathcal{L}_{\mathsf{CE}} + \mathcal{L}_{\mathsf{PCA}} + \mathcal{L}_{\mathsf{DIS}}, \text{ where } \mathcal{L}_{\mathsf{DIS}} = \mathcal{D}_{\mathsf{KL}}[g_\omega^{t-1}(f_\theta^{t-1}(x)) \| g_\omega^t(f_\theta^t(x))]_{x \sim \mathcal{P}_X^{\mathcal{T}^t}}$$

Here we have used **KL** divergence to minimize the divergence of probability distribution of previous classifier g_ω^{t-1} and current g_ω^t. Recall that KL divergence is defined as $\mathcal{D}_{\mathsf{KL}}(P\|Q) = \sum_x P(x) \log(\frac{P(x)}{Q(x)})$.

PCA

We shall use this optimization for all target domains. For the source domain specifically, there will be no KL divergence as the labels are given and PCA loss will not have the term of similarity with previous prototypes.

Referenes and Acknowledgements

- We took a little help from GPT in understanding concepts like KL divergence, RandMix.