Artificial Intelligence 356 (2026) 104544

Contents lists available at ScienceDirect

Artificial Intelligence

journal homepage: www.elsevier.com/locate/artint

Learning gradient-based mixup with extrapolation toward flatter minima for domain generalization

Danni Peng [a], Sinno Jialin Pan [b,*]

[a] Nanyang Technological University, Singapore
[b] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong, China

ARTICLE INFO

Keywords:
Domain generalization
Gradient-based mixup
Data extrapolation
Flatness-aware optimization

ABSTRACT

To address distribution shifts between training and test data, domain generalization (DG) leverages multiple source domains to learn a model that generalizes well to unseen domains. However, existing DG methods often overfit to the source domains, partly due to the limited coverage of the expected region in feature space. Motivated by this, we propose performing mixup with data interpolation and extrapolation to cover potentially unseen regions. To prevent the detrimental effects of unconstrained extrapolation, we carefully design a policy to generate the instance weights, named **F**latness-aware **G**radient-based **Mix**up (FGMix). The policy relies on gradient-based compatibilities to assign greater weights to instances that carry more invariant information and learn the mixup policy towards flatter minima for better generalization. On the DomainBed benchmark, we validate the efficacy of various designs of FGMix and demonstrate its superiority over other DG algorithms.

1. Introduction

The success of machine learning systems relies on the assumption that the training and test data are drawn from the same distribution. However, this i.i.d. assumption does not always hold in real-world applications, e.g., when the training and test data are acquired with different devices or under different conditions. When such distribution shifts occur, the systems may fail to generalize to test data if they learn to rely on spurious cues for prediction (e.g., texture or backgrounds).

Domain generalization (DG) [1–4] addresses this problem by leveraging data from multiple source domains to train a model that generalizes well to unseen target domains. Existing methods mainly focus on extracting invariant features from source domains or leveraging a meta-learning approach to learn a transferable model [2,3,5–7]. Nevertheless, most DG methods still suffer from overfitting to the source domains. As illustrated in Fig. 1, the left subfigure depicts some latent representations of data from multiple domains. The classifier is trained to perform well on the source domains (i.e., diamonds, circles, and triangles). If the target domain (i.e., stars) is located in a region that is not covered by the source domains, it is possible that the learned classifier will perform poorly on the target domain. Recently, mixup-based methods have been developed to address this issue [8–12]. Generally, interpolated data are used for model training such that the unseen regions within the convex hull of the source domains will also be covered, as shown by the solid arrows in the right subfigure of Fig. 1. But what if the target domain lies outside the convex hull? In that case, interpolated data is clearly insufficient, and one may need to consider using extrapolated data for model training (as shown by the dotted arrows).

* Corresponding author.
E-mail addresses: danni001@ntu.edu.sg (D. Peng), sinnopan@cuhk.edu.hk (S.J. Pan).

https://doi.org/10.1016/j.artint.2026.104544
Received 18 March 2023; Received in revised form 19 April 2026; Accepted 20 April 2026
Available online 23 April 2026
0004-3702/© 2026 The Author(s). Published by Elsevier B.V. This is an open access article under the CC BY-NC license (http://creativecommons.org/licenses/by-nc/4.0/).

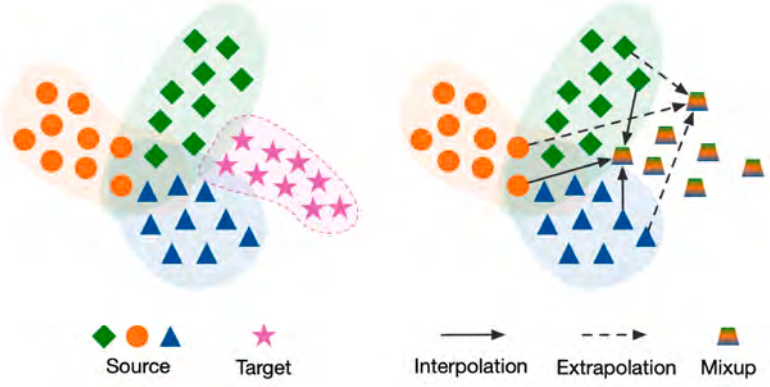


Fig. 1. Source domains, target domain, and mixup via interpolation/extrapolation.

Data extrapolation is rarely involved in the existing mixup-based methods, probably due to that without a proper mixup strategy, extrapolated data may easily deviate from the expected region and become devastating to model training. Hence, unlike existing methods that simply use random weights for mixup, a carefully designed weight-generation policy is required to produce meaningful extrapolated data. To begin with, the weight associated with each instance involved in a linear combination should be based on its relations with other instances involved in the same combination - an idea similar to context-aware attention. Inspired by the gradient-based approach for DG [5,13–15], which relies on gradient alignment for domain-invariant learning, we propose to compute the relation or compatibility score between instances based on gradients. Since the gradient compatibility indicates how much information is shared between two instances from the perspective of learning, instances with a greater sum of compatibility scores with respect to all the other instances in the same combination can be considered as carrying more invariant information and hence should be assigned greater weights [16,17]. As a result, the mixup data will absorb a larger portion of the instances containing more invariant features. To enable extrapolation, we apply a scaling and shifting operation to the normalized weights to remove the positivity constraint.

The generated mixup data is expected to encourage the learning of a more generalizable model. Hence, we propose to learn the two key components of our weight generation policy, i.e., the gradient-based compatibility scores and the extent of extrapolation, towards flatter minima of the classifier. A flat minimum is defined as a region in the loss surface where the loss varies slowly with changes in model parameters [18]. Existing flatness-aware optimization methods [18–22] aim to search for flat minima in a given loss surface (i.e., based on the original training data). Orthogonal to these approaches, we propose flattening the loss surface that the optimizer can explore. That is, we propose to learn the weight generation policy such that the resultant loss surface based on the mixture of the original and generated data is flatter. By flattening and widening the potential region the optimizer can explore, we avoid overfitting to the source domains and increase the likelihood of covering the target optima. As will be shown later, applying our approach together with an existing flatness-aware optimizer (one that seeks flatter minima on a given loss surface) further enhances performance. In addition to the flatness-aware objective, we further impose an auxiliary adversarial loss to constrain that the generated data does not deviate too much from a prior distribution as a mild regularization to prevent over-extrapolation.

To summarize, we propose a **F**latness-aware **G**radient-based **Mix**up (FGMix) method that performs mixup with gradient-based instance weights and post-processing to enable extrapolation. The learnable weight generation policy is optimized to produce a flatter loss surface for better generalization and is loosely constrained to match a prior to avoid over-extrapolation. Through extensive experiments, we validate the efficacy of various designs of FGMix quantitatively and qualitatively and show that FGMix achieves state-of-the-art performance on the DomainBed benchmark.

2. Related work

Mixup-based methods. Mixup [8] is a data augmentation method that extends the training distribution by linearly interpolating random pairs of examples and labels. Incorporating mixup data for training is equivalent to minimizing the vicinal risk [23] which enables better generalization. Recently, different forms of mix-up have been developed. Cutmix [24] cuts out a patch from an image and switches it with another image. Remix [25] assigns greater weights to the minority class label to tackle the class imbalance issue. Manifold Mixup [9] performs interpolations at the intermediate layers to enable smoothness in higher-level semantics. Motivated by the observation that the training set often contains few examples absent of the misleading spurious features, JM1 [26], and SelectMix [27] attempt to augment these examples by assigning greater weights to them in mixup, which are identified either through early stopping or with an auxiliary contrastive network. LISA [28] further considers the cross-domain scenario and performs both intra-class and intra-domain mixup, aiming to encourage the model to base its predictions solely on the domain-agnostic features while covering a broad spectrum of mixed-up features. Related to our work, MetaMixup [10] and AdaMixup [29] learn the interpolation policy adaptively from data. The former learns by simulating pseudo-targets and pseudo-sources from the actual source domains, while the latter learns to avoid the “manifold intrusion” issue caused by conflicts between mixup labels and original labels. Focusing on DG,

MixStyle [11] interpolates the feature statistics (known as styles) to synthesize novel domains. Wang et al. [12] adapt mixup to the heterogeneous setting where the label spaces are disjoint for source and target. Despite the effectiveness of various mixup methods, they primarily perform interpolation, whereas our work further explores the potential of data extrapolation to address situations where the distribution shift between source and target is significant.

Gradient-based methods. Gradients as the update steps for SGD-based optimizers lie at the heart of deep learning algorithms. However, learning a single model for multiple tasks or distributions often runs into the problem of gradient interference, which can lead to ineffective optimization [30]. In the context of DG, conflicting gradients often correspond to spurious domain-specific information, which can be detrimental to learning an invariant model [14]. The first approach to solving gradient conflicts focuses on performing some gradient surgery at each gradient step. PCGrad [31] for multi-task learning projects a task's gradient onto the normal plane of gradients of other tasks with which it has conflicts. For DG, Parascandolo et al. [13], Mansilla et al. [14] propose masking gradient components with conflicting signs across domains. Shahtalebi et al. [32] further develop a smoothed-out masking method by promoting agreement among the gradient magnitudes as well. The second approach to tackle gradient conflicts typically includes gradient alignment in the learning objective. Fish [15] explicitly optimizes the dot product between domain gradients with an efficient first-order algorithm. Fishr [33] further enforces that the variances of gradients are matched across domains. MLDG [5] employs a meta-learning approach where the meta-objective is equivalent to aligning the gradients between pseudo-source and pseudo-target domains. Different from the previous works, here we implicitly perform gradient alignment by assigning greater weights to instances whose gradients have greater overall compatibility scores to the others in the same combination. As a result, the mixup data will contain more invariant information for model learning.

Flatness-aware optimization. The connection between the flatness of minima and generalization has long been established through various theories [18,19,34,35]. In order to find a model minimizer with better generalization, existing flatness-aware methods search for flat minima in a given loss surface either by penalizing sharpness explicitly in the objective function [18–20] or performing weight averaging to reach the flatter central region of the found minima [21,22]. The latter has recently been shown to obtain remarkable results on DG tasks [36,37]. Orthogonal to these approaches, we instead propose flattening the loss surface where the optimizer can explore by generating new mixup data. This also widens the attainable minimum, increasing the likelihood of covering the target domain. We later show that, when used jointly with weight averaging to reduce variance and obtain the flatter central minimum, our method yields greater improvements.

3. Methodology

In the DG setting, suppose there are k source domains $\mathcal{S} = \{S_1, \dots, S_k\}$ available for training, and we have the training data $S_i = \{(\mathbf{x}_{i,j}, y_{i,j})\}_{j=1}^{|S_i|}$ drawn from the ith source domain S_i. Our goal is to learn a domain-invariant model $f : \mathbf{x} \to y$ from $\mathcal{S} = \{S_1, \dots, S_k\}$, which generalizes well to an unseen target domain $\mathcal{T}$. We assume the model is formed by two parts: a feature extractor g_θ and a classifier h_ϕ, i.e., $f(\mathbf{x}) = h_\phi(g_\theta(\mathbf{x}))$. Following Berthelot et al. [38], we refer to the data instances projected onto the latent space through g_θ as latent codes, i.e., the latent code of an instance $\mathbf{x}$ is $\mathbf{z} = g_\theta(\mathbf{x})$. The classifier is learned on top of the latent codes. Here, we only consider the homogeneous DG setting where the source and target domains share the same label space, i.e., $\mathcal{Y}_{S_i} = \mathcal{Y}_\mathcal{T}, \forall i \in \{1, \dots, k\}$. In this case, a single classifier h_ϕ is learned and shared across all domains.

3.1. Latent code augmentation

We perform mixup by linearly combining latent codes of the same class from different source domains. Specifically, we sample k instances $\{\mathbf{x}_1, \dots, \mathbf{x}_k\}$ with the same class label $y_1 = \dots = y_k = y$, each from one of the k source domains. We then feed the k instances into the feature extractor g_θ to obtain k latent codes $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$. Given the linear weights $\{a_1, \dots, a_k\}$ where $\sum_{i=1}^k a_i = 1$, we combine the k latent codes to obtain the mixup latent code $\mathbf{z}_{mix}$ and train the classifier h_ϕ to correctly classify $\mathbf{z}_{mix}$ to the corresponding label y. Formally, we compute the mixup latent code $\mathbf{z}_{mix}$ and its loss l_{mix} as follows:

$$\mathbf{z}_{mix} = \sum_{i=1}^{k} a_i \mathbf{z}_i, \quad l_{mix} = l(y, h_\phi(\mathbf{z}_{mix})) \tag{1}$$

To allow for both interpolation and extrapolation to occur, here we do not enforce positivity constraint on a_i. Note that when $0 \le a_i \le 1$ for all $i \in \{1, \dots, k\}$, the generated $\mathbf{z}_{mix}$ is an interpolation (i.e., within the convex hull formed by $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$), and when $a_i < 0$ for any $i \in \{1, \dots, k\}$, the generated $\mathbf{z}_{mix}$ is an extrapolation (i.e., outside the convex hull formed by $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$).

The generated latent codes are used together with the original latent codes to train the classifier h_ϕ. The feature extractor g_θ is also trained jointly to learn the domain-invariant, class-discriminative latent representations. Formally, suppose n latent codes are generated from mixup, the model parameters $\{\theta, \phi\}$ are optimized by the following objective:

$$\min_{\theta,\phi} \frac{1}{|S| + n} \left(\sum_{i=1}^{k} \sum_{j=1}^{|S_i|} l(y_{i,j}, h_\phi(\mathbf{z}_{i,j})) + \sum_{j=1}^{n} l_{mix,j} \right). \tag{2}$$

Training with mixup data helps prevent overfitting to the source domains and improves generalization of the learned classifier. Fig. 2a shows an overview of model training with the augmented data.

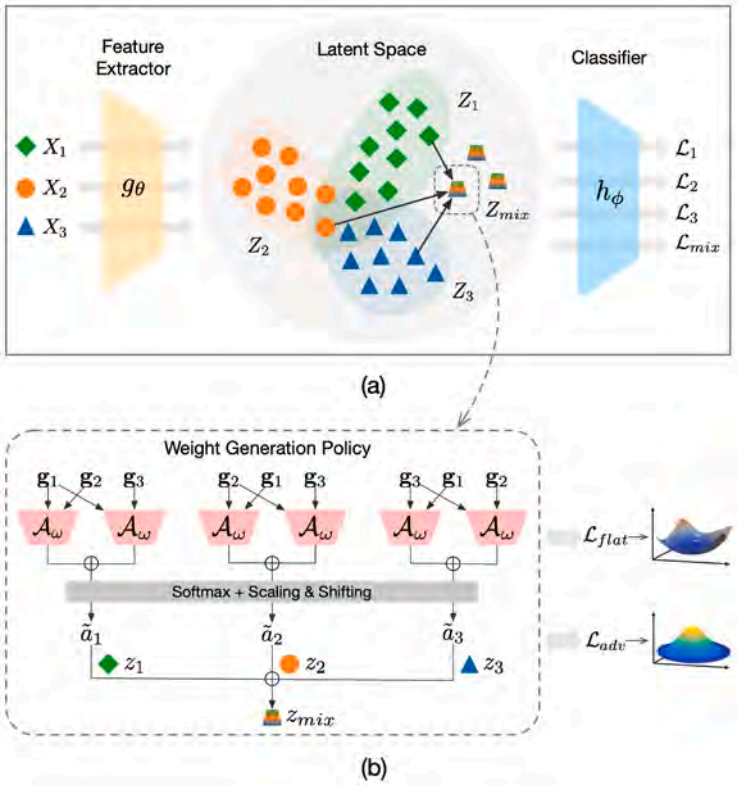


Fig. 2. Overview of the proposed FGMix. (a) Latent Code Augmentation: the mixup latent codes Z_{mix} are used together with the original latent codes $\{Z_1, Z_2, Z_3\}$ to train the classifier h_ϕ. (b) Weight Generation Policy: the weight assigned to each latent code is based on the sum of its gradient's compatibility scores w.r.t. other latent codes. A softmax and a shifting & scaling layers are applied to enable extrapolation. The compatibility score function $\mathcal{A}_\omega$ and the scaling factor λ are learned towards flatter minima (i.e., $\mathcal{L}_{flat}$) and matching the generated latent codes to a prior via adversarial training (i.e., $\mathcal{L}_{adv}$).

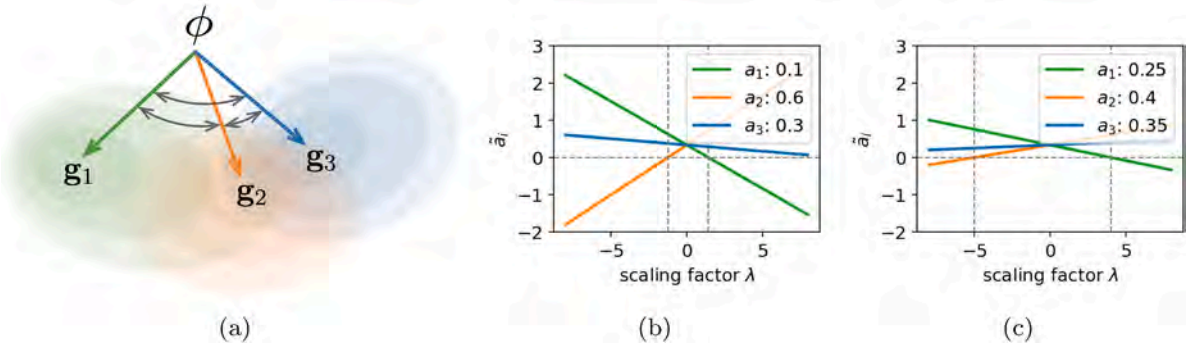


Fig. 3. Gradients of 3 instances and the effect of the scaling factor λ on the processed weight $\tilde{a}_i$ for different weight distributions. (a) shows gradients and the respective loss surfaces of z_1, z_2 and z_3 w.r.t. the classifier ϕ. (b) and (c) illustrate the effect of λ on $\tilde{a}_i$ for distinctive weight values (i.e., $a_1 = 0.1$, $a_2 = 0.6$, $a_3 = 0.3$) and similar weight values (i.e., $a_1 = 0.25$, $a_2 = 0.4$, $a_3 = 0.35$) respectively.

3.2. Gradient-based weight generation

Unlike previous methods, which sample weights $\{a_i\}_{i=1}^k$ from a pre-defined distribution [8,9,11,12], we adopt a context-aware approach that assigns weights to instances based on their relations with other instances in the same combination. While the feature-based approach specifies similarity in the latent space, it does not indicate what the classifier considers as similar. In order for the classifier to capture the invariant information, we propose to measure the similarity between instances using gradients.

Gradient similarity quantifies the level of information sharing between instances during model learning. Leveraging gradient similarity to assign weights is not new and has been adopted by several existing works in related fields [13,14,16,17,31]. Here, we

illustrate this concept with a toy example. Let $\mathbf{g}_i$ denote the gradient of the ith instance loss $l(y_i, h_\phi(\mathbf{z}_i))$ w.r.t. the classifier ϕ, i.e., $\mathbf{g}_i = \frac{\partial l(y_i, h_\phi(\mathbf{z}_i))}{\partial \phi}$. We consider 3 gradients $\{\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3\}$ of instances $\{\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3\}$ from 3 different domains as well as their respective loss surfaces, as shown in Fig. 3a. Since $\mathbf{g}_2$ and $\mathbf{g}_3$ are pointing towards similar directions (i.e., $\cos \psi_{2,3} > 0$, where $\psi_{2,3}$ is the angle between $\mathbf{g}_2$ and $\mathbf{g}_3$), taking a step along $\mathbf{g}_2$ or $\mathbf{g}_3$ will improve the classifier's performance (i.e., reducing loss) for both $\mathbf{z}_2$ and $\mathbf{z}_3$. This implies that $\mathbf{z}_2$ and $\mathbf{z}_3$ contain some shareable/invariant information as recognized by the classifier. Conversely, for $\mathbf{g}_1$ and $\mathbf{g}_3$, since they are pointing towards different directions (i.e., $\cos \psi_{1,3} < 0$), gradient update on one will degrade the classifier's performance (i.e., increasing loss) on the other, as the level of information sharing between $\mathbf{z}_1$ and $\mathbf{z}_3$ is low.

In this three-example mixup, we want to determine the relative importance of each example to assign weights. From Fig. 3a, we see that $\mathbf{g}_2$ leads to a minimum where both the loss surfaces of $\mathbf{z}_1$ and $\mathbf{z}_3$ exhibit a low value, while for $\mathbf{g}_1$, it leads to a region where none of the minima of $\mathbf{z}_2$ or $\mathbf{z}_3$ overlaps with it. If the loss value indicates how well a latent code is learned by the classifier, the fact that learning $\mathbf{z}_2$ alone decreases loss for both $\mathbf{z}_1$ and $\mathbf{z}_3$ implies that $\mathbf{z}_2$ contains some sharable/invariant information acquiring which helps improve performance for both $\mathbf{z}_1$ and $\mathbf{z}_3$. Also, we observe that $\mathbf{g}_2$ possesses the greatest sum of similarities (i.e., smallest sum of angles) with respect to the other gradients (i.e., $\mathbf{g}_1$ and $\mathbf{g}_3$), while $\mathbf{g}_1$ contains the smallest sum of similarities. Thus, we can quantify the amount of invariant information carried by an instance by the sum of similarities of the instance's gradient w.r.t. all other instances, e.g., $s_i = \sum_{j \neq i} \cos \psi_{i,j}$, which can be used as a criterion to assign weights. When an instance exhibits high aggregate gradient similarity across domains, emphasizing it in mixup causes the gradients of the generated samples to point in directions that are consistent across domains. This consistency reduces gradient conflict and stabilizes the effective update directions, biasing optimization toward a smoother loss surface with lower curvature around the classifier parameters [13,15]. Appendix A illustrates an analogy between our method of quantifying invariant information and a state-of-the-art DG method.

However, using a fixed, pre-defined measure (such as cosine similarity in the above example) to quantify similarity cannot guarantee that it yields the most suitable values for computing the mixup weights. To allow for more flexibility for the weight generation policy to generate the desired distribution, we employ a learnable function $\mathcal{A}_\omega(\cdot, \cdot)$ to measure the similarity between gradients. Specifically, we compute the sum of gradient similarities s_i (also referred to as score) for the ith latent code $\mathbf{z}_i$ as:

$$s_i = \sum_{j \neq i, j \in \{1,\dots,k\}} \mathcal{A}_\omega(\mathbf{g}_i, \mathbf{g}_j). \tag{3}$$

In practice, we use a 2-layer MLP to model $\mathcal{A}_\omega$, where the two gradients $\mathbf{g}_i$ and $\mathbf{g}_j$ are concatenated in order (i.e., $\mathbf{g}_i$ followed by $\mathbf{g}_j$) and fed into the network, akin to similarity formulations adopted in existing metric- and relation-learning models [39,40]. Note that this similarity function is asymmetric, i.e., $\mathcal{A}_\omega(a, b) \neq \mathcal{A}_\omega(b, a)$. Hence, even when we have only 2 source domains, the 2 latent codes may be assigned different weights. Also note that since $\mathcal{A}_\omega$ is a learned function (with learning objectives detailed in Section 3.3), it is not guaranteed to behave exactly like a geometric similarity metric. Hence, in what follows, we refer to its output as a *compatibility score* rather than a similarity score to avoid confusion. A discussion on the connection between the compatibility score function $\mathcal{A}_\omega$ and cosine similarity is provided in Appendix B. Here we only require the per-example gradients w.r.t. the classifier h_ϕ, which is usually shallow and close to the outputs. When applied with parallelism, e.g., using off-the-shelf vectorization functions,[1] the per-example gradient computation only incurs negligible costs.

After that, a softmax layer is applied on top of the score s_i so that the resultant weights a_i sum to 1:

$$a_i = \frac{\exp(s_i)}{\sum_{i'=1}^{k} \exp(s_{i'})}. \tag{4}$$

Since the softmax normalization produces $a_i \in (0, 1)$, this corresponds to generating $\mathbf{z}_{mix}$ as an interpolation of $\{\mathbf{z}_1, \dots, \mathbf{z}_k\}$. To enable extrapolation, we further introduce a scaling and shifting procedure applied to a_i, which lifts the positivity constraint and allows the weight to exceed 1. Specifically, we introduce a scaling factor λ and a shifting factor $\frac{\lambda - 1}{k}$ to process the weight a_i by:

$$\tilde{a}_i = \lambda a_i - \frac{\lambda - 1}{k}. \tag{5}$$

Note that the constraint $\sum_{i=1}^{k} \tilde{a}_i = 1$ is still fulfilled after the processing. The effect of varying the scaling factor λ on the processed weight $\tilde{a}_i$ is shown in Fig. 3b and c. We can see that the level of weight adjustment is controlled by the magnitude of λ, i.e., $|\lambda|$. For a more distinctive weight distribution (as indicated by 3b), the effect of scaling and shifting is more significant, and extrapolation occurs at smaller $|\lambda|$ (note that extrapolation occurs when $\tilde{a}_i < 0$ for any i). We further allow λ to be learnable so that the extent of extrapolation can be automatically controlled towards generating a desired mixup distribution. The mixup latent code $\mathbf{z}_{mix}$ is now generated by $\mathbf{z}_{mix} = \sum_{i=1}^{k} \tilde{a}_i \mathbf{z}_i$. Fig. 2b depicts the overall weight generation policy.

3.3. Learning the weight generation policy

To learn the two key components of our weight generation policy - the compatibility score function $\mathcal{A}_\omega$ and the scaling factor λ, we first encourage the generated mixup distribution to produce a flatter loss surface in the neighborhoods of the classifier ϕ. This enlarges the potential region for the optimizer to explore, thereby widening the attainable source minimum and increasing the likelihood that it covers the target optimum. Generally, the flatness of the loss surface can be measured by the Hessian-based quantities [19,34,41]

[1] https://pytorch.org/functorch/stable/functorch.html.

or Monte-Carlo approximations of the loss value in the model's neighborhoods [20,36]. For computational efficiency, we adopt the Monte-Carlo approach and measure flatness by the loss difference between the current classifier ϕ and its neighborhoods ϕ' within γ distance. Formally, we optimize the weight generation policy ω and λ with the flatness-aware loss $\mathcal{L}_{flat}$:

$$\min_{\omega,\lambda} \mathbb{E}_{\|\phi-\phi'\|\leq\gamma}[\triangle\mathcal{L}^2_{mix}], \tag{6}$$

where $\triangle\mathcal{L}_{mix} = \frac{1}{n}\sum_{j=1}^{n} l(y_j, h_\phi(\mathbf{z}_{mix,j})) - l(y_j, h_{\phi'}(\mathbf{z}_{mix,j}))$. In practice, we approximate the expectation by sampling 100 directions from a unit sphere.

To mitigate the undesirable effects of over-extrapolation, we further impose an auxiliary adversarial loss to constrain the generated distribution so that it does not deviate substantially from the prior. Similar to Li et al. [3], we match the generated distribution to a prior via adversarial training. Specifically, a discriminator $d(\cdot)$ is introduced to distinguish the generated latent codes from the ones sampled from the prior distribution. Our weight generation policy will learn to fool the discriminator into believing that the generated codes are from the prior. The adversarial loss $\mathcal{L}_{adv}$ is formulated as:

$$\min_{\omega,\lambda}\max_{d} \mathbb{E}_{\hat{\mathbf{z}}_{mix}\sim p(\hat{\mathbf{z}}_{mix})}[\log d(\hat{\mathbf{z}}_{mix})] + \mathbb{E}_{(\mathbf{z}_1,\ldots,\mathbf{z}_k)\sim p_{source}(\mathbf{z})}[\log(1 - d(\sum_{i=1}^{k}\tilde{a}_i\mathbf{z}_i))], \tag{7}$$

where $p(\hat{\mathbf{z}}_{mix})$ is a pre-defined prior distribution. In theory, we can use any arbitrary distribution for the prior. Here, we employ a Gaussian whose mean μ and variance σ^2 are computed from the source domains, i.e., $\mu = \frac{1}{|S|}\sum_{i=1}^{|S|}\mathbf{z}_i$ and $\sigma^2 = \frac{1}{|S|}\sum_{i=1}^{|S|}(\mathbf{z}_i - \mu)\circ(\mathbf{z}_i - \mu)$, where $\circ$ denotes the Hadamard product (i.e., only the diagonal entries of the covariance matrix are used). The adversarial loss is computed by sampling $\hat{\mathbf{z}}_{mix}$ from a Gaussian with μ and σ^2. In practice, since the feature extractor (and hence the latent distribution) evolves over the course of training, we compute moving averages of the mean and variance of the mini-batch to approximate μ and σ^2. In practice, we implement $d(\cdot)$ as a two-layer MLP that takes a latent code as input and outputs a probability indicating whether the code is sampled from the prior distribution or generated by the mixup module. The discriminator is trained using a standard binary cross-entropy objective to distinguish "real" latent codes drawn from the source-domain prior from generated mixup latent codes, which are treated as "fake" samples. Specifically, the discriminator is optimized by maximizing $\log d(\mathbf{z}_{real}) + \log(1 - d(\mathbf{z}_{fake}))$, while the generator is optimized to fool the discriminator by minimizing $\log(1 - d(\mathbf{z}_{fake}))$, thereby constituting an adversarial training procedure.

Here, adversarial training serves only as a mild constraint to prevent the generated mixup distribution from deviating substantially from the source distribution. It is less challenging to train than its more common uses, such as fine-grained image generation, which involve modeling complex mappings in high-dimensional image spaces and adhering to stricter learning criteria (e.g., realism and diversity) [42]. Here, the generator (i.e., the weight generation policy) only needs to produce latent codes that mimic the prior distribution in the low-dimensional latent space [43], and our application in FGMix further loosens the objective: as long as the generated distribution does not stray too far from the source prior, our target is fulfilled. In practice, we found that when using an adaptive optimizer such as Adam, we can achieve satisfactory results with minimal hyperparameter tuning effort.

Overall, we introduce a hyper-parameter α to balance the contributions of $\mathcal{L}_{adv}$ and $\mathcal{L}_{flat}$. The weight generation policy is optimized towards the following combined objective:

$$\mathcal{L}_{pol} = (1 - \alpha)\mathcal{L}_{flat} + \alpha\mathcal{L}_{adv}. \tag{8}$$

The weight generation policy is jointly learned with the base model f, i.e., we optimize (8) simultaneously with (2). To ensure that the feature extractor is well-trained and produces meaningful latent codes for mixup, learning of ω and λ (as well as the addition of the generated latent codes for classifier training) will only commence at the later stage of training (i.e., starting from the τth iteration). The overall training procedure is summarized in Algorithm 1.

3.4. Complexity analysis

Our FGMix introduces two additional modules on top of the base model, i.e., the compatibility score function $\mathcal{A}_\omega$ and the discriminator d, both adopting a lightweight architecture (i.e., a 2-layer MLP). The compatibility score function $\mathcal{A}_\omega$ takes in the concatenation of two gradients w.r.t. the classifier as input. Since $\mathcal{A}_\omega$ is asymmetric and the gradients are concatenated in order (i.e., the target gradient followed by the gradient it is compared with, as described in (3)), the compatibility score computation with $\mathcal{A}_\omega$ needs to be performed $k(k-1)$ times for a mixup of k examples, where k also refers to the number of source domains. This results in $\mathcal{O}(k^2)$ complexity for computing the mixup weights. Since, in real-world scenarios, the number of source domains k is not too large (e.g., 3 to 5 source domains in our experimental datasets), this is not a significant issue. In cases with a large number of domains, we can always adopt the stochastic approach to sample $k' < k$ domains for mixup each time to control the computational cost. Once we obtain the mixup code $\mathbf{z}_{mix}$, computing the flatness-aware loss in (6) and the adversarial loss in (7) requires only constant complexity related to the size of the discriminator and the classifier, both of which are small in scale. Also note that FGMix is applied only in the later stages of training. Hence, the overall costs incurred are considered minimal as compared to the base model training.

Algorithm 1: Training procedure of FGMix.

Required: Source data $S = \{S_1, \dots, S_k\}$; Total number of iterations T; Iteration to start training and applying the weight generation policy τ; Number of mixup instances generated in each iteration n; Learning rate of base model η_1; Learning rate of weight generation policy η_2; Neighbourhood size γ; Loss weightage α

Output: Learned feature extractor $g_\theta(\cdot)$ and classifier $h_\phi(\cdot)$.

Randomly initialize all learnable parameters.
for $t \in \{1, \dots, T\}$ **do**
  Sample a mini-batch $B = \{B_1, \dots, B_k\}$ from k source datasets $\{S_1, \dots, S_k\}$.
  if $t \geq \tau$ **then**
    for $j \in 1, \dots, n$ **do**
      Sample $(\mathbf{x}_j, y_j) \in B_i$ for $i = 1, \dots, k$ where $y_1 = \dots = y_k$.
      Obtain latent code $\mathbf{z}_i = g_\theta(\mathbf{x}_i)$ for $i = 1, \dots, k$.
      Compute linear weights $\{\tilde{a}_i\}_{i=1}^k$ by (3)–(5).
      Compute $\mathbf{z}_{mix,j}$ and its loss $l_{mix,j}$ by (1).
    Compute base model loss $\mathcal{L}_{base} = \frac{1}{|B|+n}\left(\sum_{i=1}^k \sum_{j=1}^{|B_i|} l(y_{i,j}, h_\phi(g_\theta(\mathbf{x}_{i,j}))) + \sum_{j=1}^n l_{mix,j}\right)$.
    Compute weight generation policy loss $\mathcal{L}_{pol} = (1-\alpha)\mathbb{E}_{\|\phi-\phi'\|\leq\gamma}[\triangle\mathcal{L}_{mix}^2] + \alpha[\frac{1}{n}\sum_{j=1}^n \log(1 - d(\mathbf{z}_{mix,j}))]$.
    Compute discriminator loss $\mathcal{L}_d = -\alpha[\frac{1}{n}\sum_{j=1}^n (\log d(\hat{\mathbf{z}}_{mix,j}) + \log(1 - d(\mathbf{z}_{mix,j})))]$.
    Update weight generation policy and discriminator $\omega \leftarrow \omega - \eta_2 \nabla_\omega \mathcal{L}_{pol}$, $\lambda \leftarrow \lambda - \eta_2 \nabla_\lambda \mathcal{L}_{pol}$, $d \leftarrow d - \eta_2 \nabla_d \mathcal{L}_d$.
  else
    Compute base model loss $\mathcal{L}_{base} = \frac{1}{|B|}\sum_{i=1}^k \sum_{j=1}^{|B_i|} l(y_{i,j}, h_\phi(g_\theta(\mathbf{x}_{i,j})))$.
  Update base model $\theta \leftarrow \theta - \eta_1 \nabla_\theta \mathcal{L}_{base}$, $\phi \leftarrow \phi - \eta_1 \nabla_\phi \mathcal{L}_{base}$.

4. Experiments

4.1. Experimental setup

We conduct experiments mainly on DomainBed [44], a recently introduced testbed that provides a unified evaluation procedure for DG algorithms. Following the previous DG studies [36,37], we focus on five real-world benchmark datasets available on DomainBed: `PACS` [4] (4 domains, 7 classes, and 9991 images), `VLCS` [45] (4 domains, 5 classes, and 10,729 images), `OfficeHome` [46] (4 domains, 65 classes, and 15,588 images), `TerraIncognita` [47] (4 domains, 10 classes, and 24,788 images) and `DomainNet` [48] (6 domains, 345 classes, and 586,575 images).

For a fair comparison, we follow the experimental settings by Gulrajani and Lopez-Paz [44], including data splits (20% data are reserved for validation for each training domain), hyper-parameter search (a search distribution is pre-defined for each hyper-parameter), number of iterations (default to be 15,000 for `DomainNet` and 5000 for the other datasets), image augmentation (cropping, resizing, horizontal flips, color jitter, grayscaling, normalization, etc.) and the base model backbone (ResNet-50 [49] pre-trained on ImageNet as initialization).

For our proposed FGMix, the number of instances k used for mixup is set to the number of source domains in each dataset ($k = 3$ for `PACS`, `VLCS`, `OfficeHome`, and `TerraInc`; $k = 5$ for `DomainNet`), with one instance sampled from each source domain. We set the iteration τ to start training and applying weight generation for mixup as 9000 for `DomainNet` and 3000 for the other datasets and set the size of mixup data n generated in each iteration as 32, same as the batch size for each source domain.[2] We tune the learning rate of weight generation policy and the discriminator in {1e-4, 1e-3, 1e-2}, the initialization of the scaling factor λ in {1, 3, 5, 8},[3] the neighborhood size γ in $10^{\text{Uniform}(0,2)}$, and the loss weightage α in {0.01, 0.03, 0.1, 0.3} on the training domain validation sets. All experiments are repeated for 5 trials with different random seeds.[4] Regarding the network architecture, we use a 2-layer MLP for the compatibility score function $\mathcal{A}_\omega$ accompanied with tanh activations. The hidden sizes for both layers are set to 64. The discriminator d is also a 2-layer MLP with a hidden size 64.

4.2. Baseline methods and hyperparameter settings

We select several baselines related to our method for overall comparison on DomainBed[5]: (1) naive baseline without any DG strategy: **ERM** [51] (empirical risk minimization); (2) mixup-based methods: **Mixup** [12] (mixup at the input level), **Manifold Mixup** [9] (mixup at the feature level), **MixStyle** [11] (mixup of feature statistics) and **MetaMixup** [10] (meta-learn the interpolation policy); (3) gradient-based methods: **PCGrad** [31] (project gradients onto the normal plane of conflicting gradients), **AND-Mask** [13] (mask out conflicting gradient components), **Fish** [15] (optimize for gradient alignment) and **Fishr** [33] (optimize for gradient covariance

[2] Note that since `DomainNet` has large label size (345 classes in total), applying the default random batch sampler in DomainBed cannot ensure that there are enough same-class examples in each batch for mixup. Hence, we modify the batch sampler by first sampling m classes ($m \ll$ batch size) and then drawing examples from the m classes to form the mini-batch for each source domain.

[3] Note that $\lambda = 1$ is equivalent to having no scaling & shifting effect.

[4] Experiments are conducted on NVIDIA A100 with 40GB memory.

[5] We note that ERM++ [50] is also evaluated on DomainBed but employs several modifications to the standard setup. Hence, it is not included here for a fair comparison.

Table 1
Overall comparison of selected algorithms on five datasets. Model selection is based on **training domain validation**. The reported result for each dataset is the average performance across all test domains for that dataset. Most of the results are obtained directly from the literature, except for those denoted with †, which are from our reproduction on DomainBed. We also report the time cost (in ms) per training iteration, averaged over 5000 iterations. The time reported is for datasets with 3 source domains, and thus uses $k = 3$ instances for mixup each time.

	Algorithm	PACS	VLCS	OfficeHome	TerraInc	DomainNet	Avg.	Time
	ERM [51]	85.5±0.2	77.5±0.4	66.5±0.3	46.1±1.8	40.9±0.1	63.3	284.8
Mixup-based Methods	Mixup [12]	84.6±0.6	77.4±0.6	68.1±0.3	47.9±0.8	39.2±0.1	63.4	285.6
	Manifold Mixup† [9]	86.2±0.6	76.7±1.1	67.3±1.0	48.8±2.1	41.2±0.3	64.0	444.5
	MixStyle† [11]	85.3±1.9	77.4±0.8	67.3±0.9	46.8±1.1	40.9±0.2	63.5	290.2
	MetaMixup† [10]	85.2±1.2	77.9±0.8	68.2±0.7	47.1±1.8	41.8±0.3	64.0	750.7
Gradient-based Methods	PCGrad† [31]	85.0±0.9	77.5±0.8	65.5±0.9	48.3±2.3	41.1±0.2	63.5	310.0
	AND-Mask [13]	84.4±0.9	78.1±0.9	65.6±0.4	44.6±0.3	37.2±0.6	62.0	319.4
	Fish [15]	85.5±0.3	77.8±0.3	68.6±0.4	45.1±1.3	**42.7**±0.2	63.9	671.7
	Fishr [33]	85.5±0.4	77.8±0.1	67.8±0.1	47.4±1.6	41.7±0.0	64.0	703.3
Augmentation Methods	L2A-OT† [52]	85.8±1.9	77.4±0.9	68.1±1.6	48.6±2.1	40.2±0.5	64.0	622.7
	CNSN† [53]	85.6±0.9	77.1±1.0	67.3±1.2	48.4±1.6	40.7±0.4	63.8	412.1
	DDG† [54]	85.3±1.6	76.8±1.2	68.1±1.0	47.7±2.1	40.0±0.5	63.6	653.2
DomainBed SOTA	SelfReg [56]	85.6±0.4	77.8±0.9	67.9±0.7	47.0±0.3	41.5±0.2	64.0	320.1
	SagNet [55]	86.3±0.2	77.8±0.5	68.1±0.1	48.6±1.0	40.3±0.1	64.2	671.7
	CORAL [57]	86.2±0.3	**78.8**±0.6	68.7±0.3	47.6±1.0	41.5±0.1	64.6	305.8
	FGMix (ours)	**86.8**±1.2	78.1±0.9	**69.0**±1.3	**49.2**±1.4	42.1±0.5	**65.0**	312.7
Combined with flatness-aware optimizer SWA [21,37]								
	ERM + SWA†	87.0±0.5	77.2±0.6	69.5±0.4	50.1±0.7	44.0±0.2	65.6	293.1
	SelfReg + SWA	86.5±0.3	77.5±0.0	69.4±0.2	51.0±0.4	44.6±0.1	65.8	327.1
	CORAL + SWA†	87.5±0.5	78.2±0.4	70.7±0.1	51.1±0.6	44.6±0.4	66.4	314.2
	FGMix + SWA	**88.4**±0.6	**78.7**±0.6	**71.3**±0.6	**52.3**±0.9	**45.1**±0.4	**67.2**	319.8

alignment); (4) augmentation methods: **L2A-OT** [52] (generate pseudo-source by enlarging divergence to the source domains), **CNSN** [53] (exchange and normalize instances' styles), **DDG** [54] (disentangle and swap instances' variation factors) (5) current SOTA on DomainBed: **SagNet** [55] (reduce style bias), **SelfReg** [56] (self-supervised contrastive regularization) and **CORAL** [57] (correlation alignment).

For our FGMix and all the reproduced algorithms (except for DDG, which will be detailed later), we train the base model by setting the batch size as 32 (due to constraints in computational resources) and tuning the dropout rate in {0, 0.1, 0.5}, learning rate in {1e-5, 3e-5, 5e-5} and weight decay in {1e-4, 1e-6}, following Cha et al. [36].

We reproduced 7 baselines (i.e., Manifold Mixup [9], MixStyle [11], MetaMixup [10], PCGrad [31], L2A-OT [52], CNSN [53] and DDG [54]) due to their lack of results on DomainBed benchmark [44]. We directly adopt the official implementations released by the respective authors in the DomainBed environment (refer to the appendix of Gulrajani and Lopez-Paz [44] for how to incorporate new algorithms into DomainBed).

For Manifold Mixup, we set $\alpha = 0.2$ for the beta distribution from which the interpolation constant is sampled. For MixStyle, we set $\alpha = 0.1$ for the Beta distribution and $p = 0.5$ for the probability of applying MixStyle. As recommended, we insert the MixStyle layer after the 1st and 2nd residual blocks. For MetaMixup, we set the learning rate for the interpolation constant to 1e-3 (similar to our weight generation policy). PCGrad is free of additional hyperparameters. For L2A-OT, we use ResNet-50 as both the classifier and the domain discriminator to compute the domain divergence. Following the suggestions by the authors, we search λ_{Domain} in {0.5, 1, 2}, λ_{Cycle} in {10, 20} and λ_{CE} in {1}. For CNSN, we apply both CrossNorm (2-instance mode) and SelfNorm, inserting them at the end of each residual block in ResNet-50. We set the number of active CrossNorm layers to 1 and the probability of applying CrossNorm to 0.5 to avoid over-augmentation. For DDG, a 2-stage procedure is adopted, where the generator is pre-trained as part of a GAN in the first stage and then applied (and further updated) in the second stage. Following the official implementation, the batch size is set to 2, and the number of training steps is enlarged to 25,000 for the first stage and 10,000 for the second stage. Due to the 2-stage training and the larger number of training steps, DDG takes much longer to train than other baselines, making it less competitive.

To apply SWA, we follow Arpit et al. [37], which suggests skipping the first few iterations and starting to apply weight averaging from the 100th iteration.

Table 2
Ablation study on PACS and TerraInc. From A to FGMix we add one component at a time, where A is a simple interpolation baseline with random instance weights. For Variant F, we replace the gradient-based inputs in FGMix with latent features.

Variant	similarity-based weights	gradient-based similarity	scaling & shifting	$\mathcal{L}_{flat}$	$\mathcal{L}_{adv}$	PACS	TerraInc
A (baseline)						84.7±0.7	47.5±1.1
B	✓					85.2±0.5	47.9±0.9
C	✓	✓				85.6±0.2	48.1±1.1
D	✓	✓	✓			86.0±1.1	48.8±1.7
E	✓	✓	✓	✓		86.7±1.7	**49.2**±2.2
F	✓		✓	✓	✓	86.2±1.5	48.9±1.7
FGMix (ours)	✓	✓	✓	✓	✓	**86.8**±1.2	**49.2**±1.4

4.3. Overall comparison

The comparison results with baselines on the 5 datasets are presented in Table 1 (see Appendix C.1 for the disentangled results on each test domain in Tables C.1–C.5). We observe that most of the DG algorithms outperform ERM in terms of average accuracy (except for AND-Mask). Our proposed FGMix achieves consistent performance gains over ERM for all 5 datasets: +1.3pp for PACS, +0.6pp for VLCS, +2.5pp for OfficeHome, +3.1pp for TerraInc and +1.2pp for DomainNet. We observe that FGMix performs exceptionally well on the more challenging datasets, where distribution shifts between source and target domains may be more pronounced, i.e., the gains are most pronounced for OfficeHome and TerraInc, whose test accuracies are relatively low compared to other datasets. Overall, our FGMix tops in 3 out of 5 datasets, achieving the best average accuracy with 0.4pp higher than the previous SOTA (i.e., CORAL), and 1.0pp higher than the strongest relevant baselines (i.e., Manifold Mixup, MetaMixup, Fishr, and L2A-OT).

We further conduct experiments to combine FGMix with the flatness-aware optimizer SWA (Stochastic Weight Averaging) [21,37]. SWA simply performs weight averaging, yielding a minimizer in the flatter central region of the found minimum. A recent study by Arpit et al. [37] found that SWA not only boosts the performance of DG algorithms but also ensures a better correlation between in-domain validation and out-domain test results, facilitating more reliable model selection. We combine SWA with FGMix and 3 other strong and representative DG algorithms (i.e., ERM, SelfReg, and CORAL) for comparison. Table 1 shows that FGMix + SWA achieves the best results for all 5 datasets. Notably, we observe that the performance gain is most pronounced for FGMix after combining with SWA, compared to SelfReg and CORAL (i.e., +1.8pp for SelfReg and CORAL, and +2.2pp for FGMix after combining with SWA). This shows that FGMix is more orthogonal to SWA than other DG methods, as its benefits persist even when combined with SWA. To understand this, recall that FGMix is used to widen the loss surface for the optimizer to explore. While this increases the chance of covering the target area, it also enlarges the off-target area reachable to the optimizer. SWA with weight averaging along the training process helps mitigate the risk of the optimizer running into an undesirable region.

In Table 1, we also report the time cost (in ms) per training iteration averaged over 5000 iterations. The reported time is for datasets with 3 source domains and thus uses $k = 3$ instances for mixup each time. We observe that the average per-iteration time cost for FGMix is only slightly higher than that of ERM. Among the competitive baselines (e.g., DomainBed SOTA), FGMix's time cost is on par with SelfReg and CORAL, and much lower than that of SagNet. Notably, when used with a flatness-aware optimizer such as SWA, the performance gain of FGMix becomes more pronounced, while the time cost remains the same for FGMix and other SOTA methods. Since the inference time cost of FGMix is the same as that of ERM (i.e., FGMix performs inference using only the trained base model), the minor additional costs incurred during training are considered acceptable given the evident performance gains.

4.4. Ablation study

We test the efficacy of various components of FGMix by introducing 5 variants, each includes one additional component at a time: A is a simple interpolation baseline with random instance weights drawn from a Dirichlet distribution; B employs cosine similarity to compute the weights based on instances' latent codes (i.e., feature-based similarity); C computes cosine similarity based on latent codes' gradients w.r.t. the classifier (i.e., gradient-based similarity); D further includes scaling and shifting for extrapolation by setting the scaling factor λ as 3; E replaces the cosine similarity with a learnable compatibility score function $\mathcal{A}_\omega$ and learns both $\mathcal{A}_\omega$ and λ towards flatter loss surface with the objective $\mathcal{L}_{flat}$. Further including the adversarial loss $\mathcal{L}_{adv}$ for prior matching results in our proposed FGMix. We also include a Variant F, which replaces the gradient-based inputs in FGMix with latent features, to examine feature-based conditioning for $\mathcal{A}_\omega$.

Table 2 presents the results (see Appendix C.2 for results on each test domain in Tables C.6–C.7). From A to FGMix, the incremental gains are +0.5pp, +0.4pp, +0.4pp, +0.7pp, +0.1pp for PACS, and +0.4pp, +0.2pp, +0.7pp, +0.4pp, +0.0pp for TerraInc. Firstly, we see that enabling data extrapolation (i.e., scaling & shifting) and learning towards a flatter loss surface (i.e., $\mathcal{L}_{flat}$) seem to be the two most important components, as both serve to avoid overfitting to the source domains and increase the chance of covering the target domain. The similarity-based weight computation that accounts for attention to other instances within the same combination also plays an important role, yielding evident improvements over the random weight baseline. Replacing feature-based similarity with

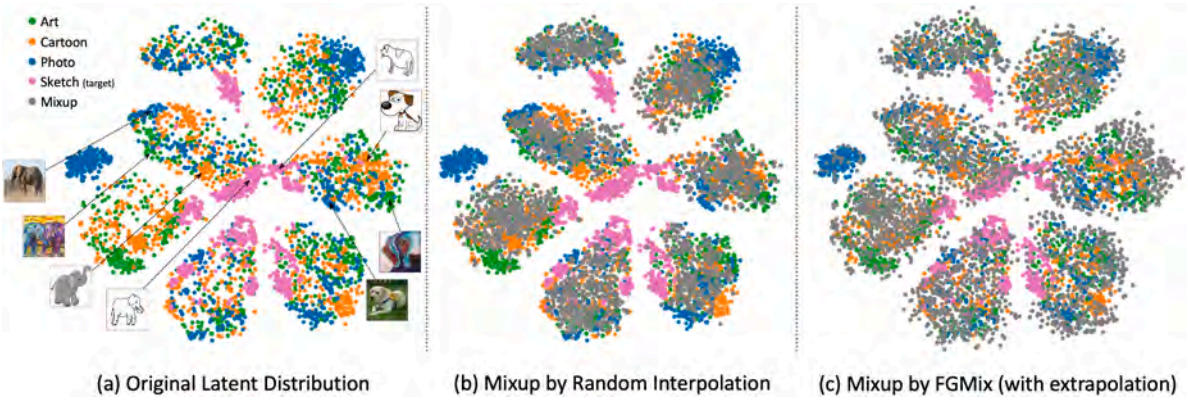


Fig. 4. t-SNE visualization of the latent distributions of PACS trained by using "Art", "Cartoon", "Photo" as the source domains and "Sketch" as the target domain. We plot the mixup distribution generated by random interpolation and by our FGMix which involves extrapolation.

gradient-based similarity also improves performance, as similarity in feature space does not characterize what the classifier deems invariant, and hence, weights based on feature similarity cannot identify instances that carry more invariant information. This trend is further confirmed by Variant F, which replaces gradient-based inputs in FGMix with latent features when learning $\mathcal{A}_\omega$. The fact that feature-based conditioning consistently underperforms FGMix highlights that gradients provide a more direct and informative signal for guiding flatness-aware mixup. Lastly, though matching the generated distribution to a prior (i.e., $\mathcal{L}_{adv}$) seem to have minor effects on the mean accuracy, it helps reduce the variance significantly from E to FGMix, demonstrating its role as a regularization measure to prevent over-extrapolation. In Appendix C.3, we include a sensitivity test to evaluate the importance of $\mathcal{L}_{adv}$ by varying the weight α.

4.5. Qualitative analysis

4.5.1. Mixup distribution visualization

To understand how the generated distribution facilitates classifier learning, we visualize the latent codes of PACS in 2D using t-SNE. For this visualization, we train a model (i.e., feature extractor + classifier) with FGMix and obtain the latent codes from the feature extractor. To compare different mixup methods, we plot the mixup distribution generated by random interpolation (i.e., variant A) and by our FGMix, which involves extrapolation. Here, we use "Sketch" as the target domain, as it is the most difficult domain when training with the other 3 domains.

Fig. 4 shows the distributions of the original latent codes (i.e., Fig. 4a), the mixup latent codes generated by random interpolation (i.e., Fig. 4b) and by FGMix (i.e., Fig. 4c), respectively. In Fig. 4a, we observe that the latent codes of the target domain "Sketch" (i.e., the pink dots) are located near the decision boundary, especially for the "dog" and "elephant" classes, which are inseparable in the middle. In Fig. 4b, the mixup codes generated by random interpolation all lie within the convex hull formed by the source latent codes in the respective cluster, leaving the target region uncovered. Our FGMix (as shown in Fig. 4c), on the other hand, is able to generate codes outside of the convex hull with extrapolation, resulting in better coverage of the target region and hence better generalization performance of the classifier learned from the mixup latent codes. In this case, approximately 22.5% of the generated mixup codes are extrapolated, and the remainder are interpolations. One possible reason why $\mathcal{L}_{flat}$ can produce difficult samples near the decision boundary is that, as training progresses, increased prediction confidence flattens the local loss landscape, which may be associated with a smoother boundary and thus allows boundary-adjacent samples to remain robust to parameter perturbations. In Appendix C.4, we further include an analysis regarding the effect of mixup on the learned latent distributions.

4.5.2. Loss landscape visualization

In this section, we verify that FGMix indeed generates flatter loss minima. We also observe how changes in the loss landscape affect the optimization process. The experiments are conducted on the TerraInc dataset.

Flatness analysis. We compare the flatness of FGMix minimizer with that of ERM, random mixup (i.e., variant A), and gradient-based similarity mixup (i.e., variant C) in Fig. 5. Following Izmailov et al. [21], Cha et al. [36], for each method, we plot the square loss difference between the minimizer ϕ^* and its neighborhoods ϕ' at distance γ, i.e., $\mathbb{E}_{\|\phi^*-\phi'\|=\gamma}[\mathcal{L}(\phi^*)-\mathcal{L}(\phi')]^2$, for $\gamma = 1, \ldots, 10$. The expectation is approximated by averaging over 50 randomly sampled directions. Note that for ERM, the loss is computed based on the original training data, while for variant A, variant C, and FGMix, the loss is computed based on the mixture of original and mixup data.

Fig. 5 shows the separate plots for each of the 4 methods (including the results of the 50 sampled directions) and the combined plots. From top to bottom, each row shows the results obtained using L100, L38, L43, and L46 as the target domain, respectively. First, by comparing variants A and C with ERM, we see that simply adding mixup data yields a sharper loss minimum. Fortunately,

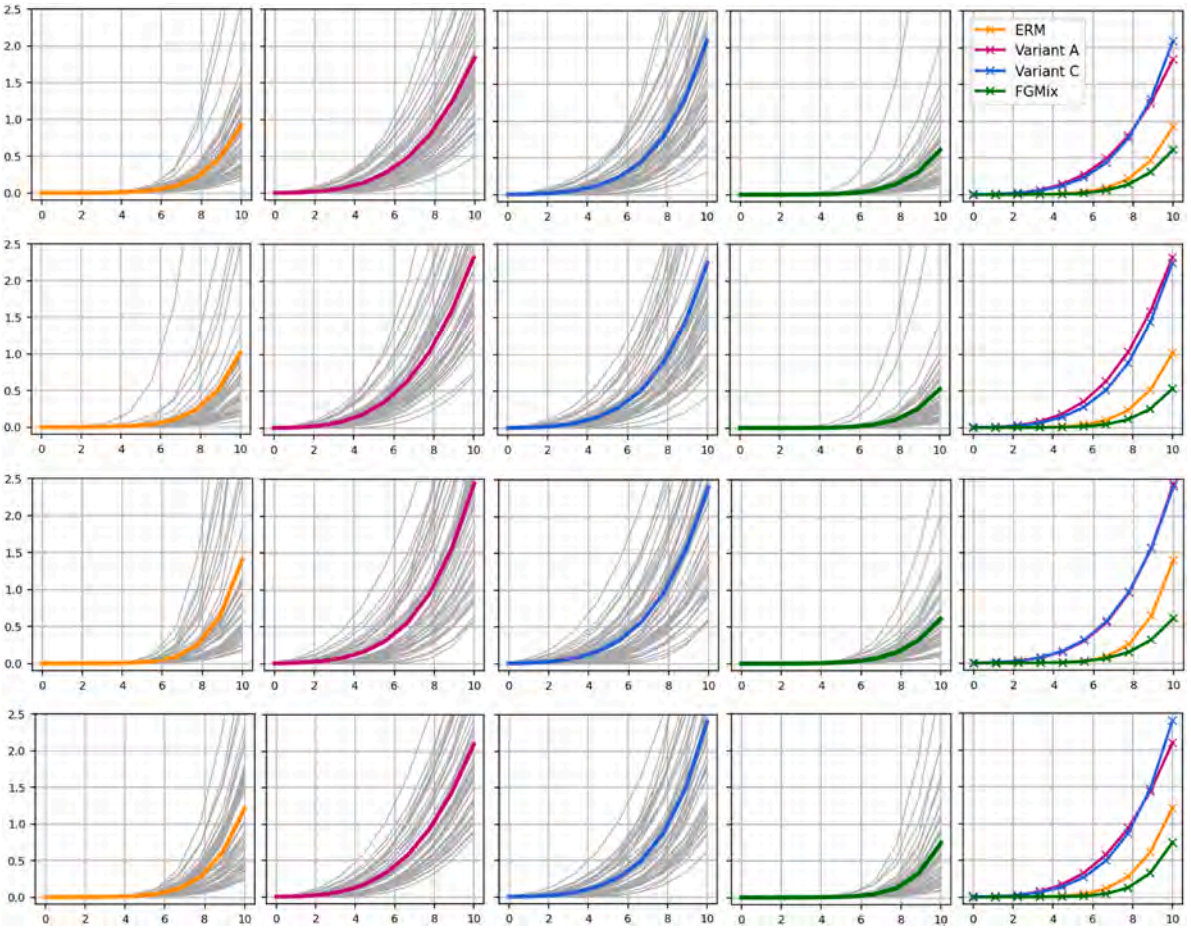


Fig. 5. Square loss difference between the classifier minimizer and its neighborhoods at distance $\gamma = 1, \ldots, 10$ for ERM, variant A, variant C, and FGMix. Each row shows the results of using one of the 4 domains as the target domain (from the top down, the target domain is L100, L38, L43, and L46, respectively). We randomly sample 50 directions to compute the loss difference, indicated by the grey curves. The final results averaged over 50 random directions are compared in the last column.

our FGMix with the flatness-aware objective effectively reduces loss sharpness by adjusting the weight generation policy, yielding a minimum that is even flatter than that of ERM.

Loss surface & optimization path. To observe how the generated data affects the optimization process, in Fig. 6, we plot the 2D loss surface of the original training data (i.e., ERM) as well as the one after adding the generated mixup data[6] (i.e., FGMix). We also plot the accuracy surface on the test data to visualize distribution shifts relative to the training domains. For the optimization paths, we consider both the original paths of ERM and FGMix and the ones with SWA, as weight averaging helps stabilize the optimization process and converge to a better minimum. To generate the 2D loss surface, we follow Izmailov et al. [21] by choosing 3 models to compute the orthonormal basis of the 2D plane containing the 3 models (computation details are included in Appendix D). For a better span of the 2D perspective, the 3 models chosen are 1) the common initialization, 2) the final checkpoint of ERM + SWA, and 3) the final checkpoint of FGMix + SWA.

Fig. 6a and b show the train loss surfaces and optimization paths for ERM and FGMix, respectively, and Fig. 6c shows the test accuracy surface with paths of both ERM and FGMix on it. From top to bottom, each row presents the results obtained using L100, L38, L43, and L46 as the target domain, respectively. First, note that, since the generated data is included in model training only from the 3,000th iteration onwards, ERM and FGMix follow the same path before the 3,000th step. Comparing Fig. 6a and c, we see that there is a clear shift in optimum between the ERM train loss and the test accuracy, which results in the ERM optimization path converging to a point far from the test optimum. While for FGMix (as shown in Fig. 6b), the loss surface is flatter as compared to that of ERM after adding the mixup data, yielding a broader minimum that better covers the test optimum. As a result, the FGMix

[6] The mixup data used here is generated by the well-trained $\mathcal{A}_\omega$ and scaling factor λ at the last iteration.

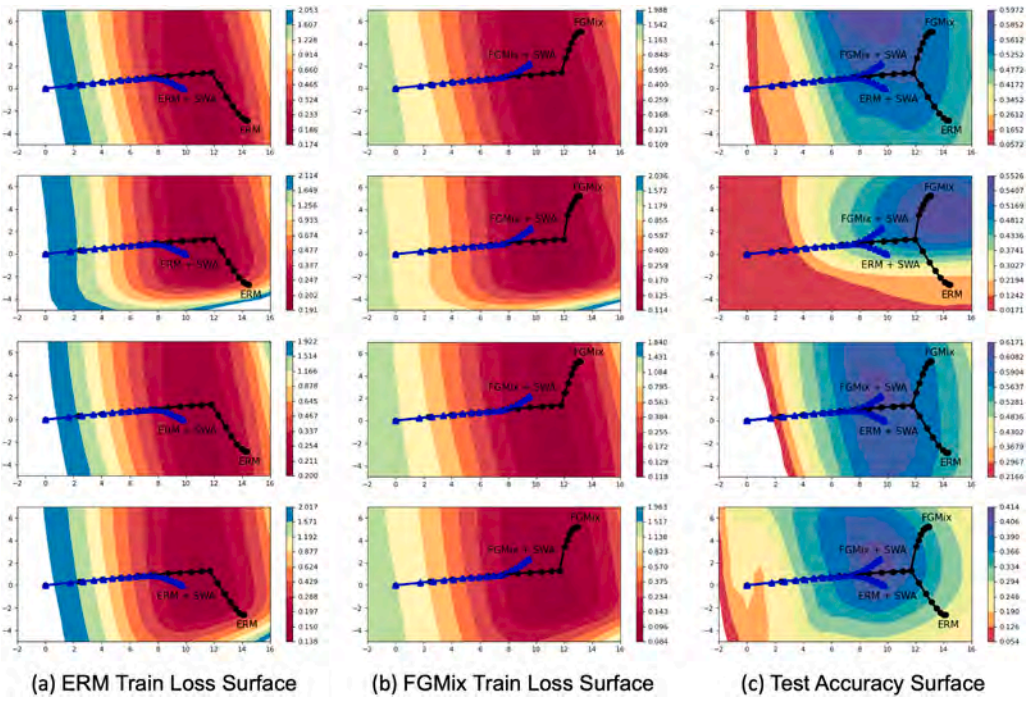


Fig. 6. 2D loss and accuracy surfaces generated using 3 models at the initialization, the end of ERM + SWA, and the end of FGMix + SWA, respectively. Each row shows the results obtained by using one of the 4 domains as the target domain (from top to bottom, the target domains are L100, L38, L43, and L46, respectively). Note that only the 3 models used to generate the plots are on the 2D plane (i.e., the loss/accuracy values reported for them are correct). Also note that for FGMix, since the mixup data is only added for training from the 3,000th iteration onwards, the loss values indicated for the optimization path before the 3,000th iteration in (b) do not reflect the actual loss during training. For direct comparison of the flatness, we use the same scale interval for (a) and (b).

optimization path converges to a point closer to the test optimum; in Fig. 6c, FGMix + SWA terminates in a region with higher test accuracy than ERM + SWA. The result is consistent across all 4 experiments with different target domains.

5. Conclusion and discussion

In this work, we explore a mixup method with data extrapolation. We propose a weight generation policy, FGMix, that computes instance weights from gradients and learns toward flatter minima. We also employ an adversarial loss for regularization. Extensive experiments on the DomainBed benchmark demonstrate FGMix's effectiveness. Perhaps the major limitation of FGMix is the large variance caused by data extrapolation, as we lack strategies to guide extrapolation toward the expected region. In the future, we will consider exploiting domain-specific information and relationships among the source domains to design a more effective extrapolation strategy.

CRediT authorship contribution statement

Danni Peng: Writing – original draft, Software, Methodology; **Sinno Jialin Pan:** Writing – original draft, Supervision, Methodology.

Data availability

Data will be made available on request.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

Sinno J. Pan thanks the support from the JC STEM Lab of Machine Learning and Symbolic Reasoning funded by The Hong Kong Jockey Club Charities Trust.

Appendix A. On the Connection with other Gradient Methods

Like our proposed method for quantifying the amount of invariant information based on the sum of gradient similarities, a few recent DG works on gradient surgery also place greater emphasis on examples whose gradients align more closely with those of other examples, aiming to capture more invariant features in the model [13,14]. In [14], the authors empirically show that the gradient cosine similarity is higher for inter-domain pairwise example comparisons as opposed to the intra-domain comparisons. They subsequently propose zeroing out or randomizing the gradient component when there is a conflicting sign across gradients between different domains. AND-Mask [13] performs a similar procedure, except that it loosens the criteria by zeroing out gradient components only when the percentage of disagreeing signs is greater than a threshold. These methods implicitly place more emphasis on more aligned gradients, as the retained gradient components are largely contributed by them, which constitute the majority of the population.

In the following, we present a toy example to illustrate the analogy between our proposed FGMix and AND-Mask [13], yielding a similar gradient update direction for learning the model. Fig. A.1 shows 4 gradients of 4 examples belonging to different domains, i.e., $\mathbf{g}_1 = [-0.5, 1], \mathbf{g}_2 = [-1, 0.5], \mathbf{g}_3 = [-1, -1], \mathbf{g}_4 = [1, -1]$. Standard empirical risk minimization (ERM) updates the model using the overall gradient $\mathbf{g}_{ERM} = \sum_i \mathbf{g}_i = [-1.5, -0.5]$. AND-Mask eliminates the gradient component when the percentage of disagreeing signs exceeds a pre-defined threshold. Suppose we set the threshold at 30%. The first gradient component has 25% disagreeing signs, i.e., $\text{sign}(\mathbf{g}_1^1) = \text{sign}(\mathbf{g}_2^1) = \text{sign}(\mathbf{g}_3^1) \neq \text{sign}(\mathbf{g}_4^1)$. The second gradient component has 50% disagreeing signs, i.e., $\text{sign}(\mathbf{g}_1^2) = \text{sign}(\mathbf{g}_2^2) \neq \text{sign}(\mathbf{g}_3^2) = \text{sign}(\mathbf{g}_4^2)$. Hence, the first component is retained, while the second component is replaced with zero, resulting in the overall gradient $\mathbf{g}_{AND} = [-1.5, 0.0]$. For our FGMix, we compute the weight associated with $\mathbf{g}_i$ based on the sum of cosine similarities $s_i = \sum_{j \neq i} \cos \psi_{i,j}$ followed by softmax normalization. The computation is shown in Table A.3. Based on the weight a_i computed, the aggregated gradient of FGMix is $\mathbf{g}_{FGMix} = [-0.793, 0.023]$.

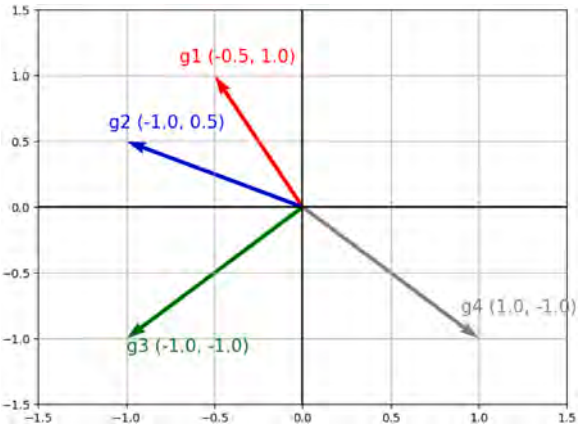


Fig. A.1. Gradients of 4 examples from different domains.

To summarize, the gradients obtained using the 3 methods are:

$$\mathbf{g}_{ERM} = [-1.5, -0.5], \quad \mathbf{g}_{AND} = [-1.5, 0.0], \quad \mathbf{g}_{FGMix} = [-0.793, 0.023].$$

By comparing $\mathbf{g}_{FGMix}$ with $\mathbf{g}_{ERM}$ and $\mathbf{g}_{AND}$, we find that the cosine similarity between $\mathbf{g}_{FGMix}$ and $\mathbf{g}_{AND}$ is higher than that between $\mathbf{g}_{FGMix}$ and $\mathbf{g}_{ERM}$, i.e., $\cos \psi_{FGMix,AND} = 0.9996$, $\cos \psi_{FGMix,ERM} = 0.9389$. This suggests that FGMix produces gradient update with direction similar to that of AND-Mask, which share the same goal of capturing invariant information for better generalizing across domains.

Table A.3
FGMix weight computation based on the sum of gradient cosine similarities.

		$\cos \psi_{i,1}$	$\cos \psi_{i,2}$	$\cos \psi_{i,3}$	$\cos \psi_{i,4}$	$s_i = \sum_{j \neq i} \cos \psi_{i,j}$	$a_i = \frac{\exp(s_i)}{\sum_{i'=1}^{k} \exp(s_{i'})}$
$\mathbf{g}_1$	[-0.5, 1]	-	0.8	-0.316	-0.949	-0.465	0.21218712
$\mathbf{g}_2$	[-1, 0.5]	0.8	-	0.316	-0.949	0.168	0.39938428
$\mathbf{g}_3$	[-1, -1]	-0.316	0.316	-	0.0	0.0	0.33777486
$\mathbf{g}_4$	[1, -1]	-0.949	-0.949	0.0	-	-1.897	0.05065374

Note that although $\mathbf{g}_{FGMix}$ and $\mathbf{g}_{AND}$ exhibit high similarity in this toy example, our actual FGMix algorithm functions differently and achieves different effects, as it employs a learnable similarity function $\mathcal{A}_\omega$ instead of the pre-defined cosine similarity function used in this example. We use cosine similarity to illustrate our motivation for simplicity; the same idea applies to other similarity metrics.

Appendix B. On the Connection between $\mathcal{A}_\omega(\cdot)$ and Cosine Similarity

As discussed in the main text, cosine similarity is introduced primarily as a motivating example to provide intuition. Since $\mathcal{A}_\omega(\cdot)$ is a learned function optimized toward the designed objectives, it is not guaranteed to behave exactly like cosine similarity or any other conventional geometric similarity metric. Instead, the role of $\mathcal{A}_\omega$ is to assign higher scores to pairs of gradients whose joint effect promotes flatter optimization landscapes under mild prior-matching regularization.

The behavior of $\mathcal{A}_\omega$ is mainly shaped by the flatness-aware loss in Eq. (6), which favors mixup samples whose induced gradients remain stable under parameter perturbations. Under such an objective, gradient pairs with lower cosine similarity or stronger misalignment are relatively less favored, as they are more likely to induce higher gradient variance, sharper curvature, and greater loss sensitivity-behaviors penalized by the flatness objective. Consequently, although $\mathcal{A}_\omega$ is unconstrained in form, the optimization pressure biases it toward assigning higher scores to more aligned gradient pairs. This leads to similarity-like behavior *in expectation*, while still allowing flexibility beyond conventional similarity measures.

To empirically examine this behavior, we analyze the relationship between the learned $\mathcal{A}_\omega(\cdot)$ scores and cosine similarity using 200 randomly sampled cross-domain gradient pairs across six cross-domain settings (excluding the target domain) on the PACS dataset. Fig. B.1 presents scatter plots of $\mathcal{A}_\omega$ versus cosine similarity for each domain pair. We observe consistent positive Spearman correlations, ranging from approximately 0.35 to 0.58 across all six settings, indicating that gradient pairs with higher cosine similarity tend to receive higher $\mathcal{A}_\omega$ scores. At the same time, some gradient pairs with low cosine similarity can still obtain high $\mathcal{A}_\omega$ scores, showing that such cases are not strictly excluded. This suggests that $\mathcal{A}_\omega$ captures additional factors beyond pure geometric similarity, consistent with its role as a learned compatibility measure rather than a fixed similarity metric.

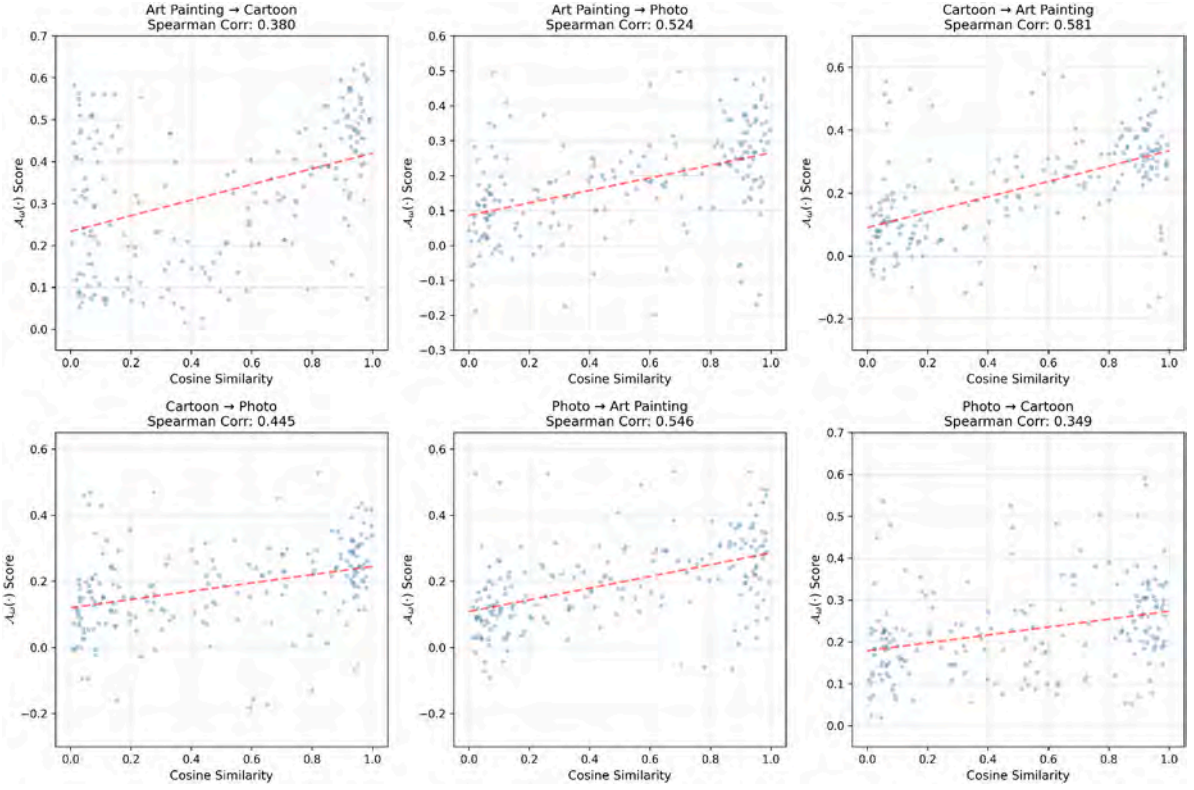


Fig. B.1. Scatter plots of learned $\mathcal{A}_\omega(\cdot)$ scores versus cosine similarity for 200 randomly sampled cross-domain gradient pairs across six cross-domain settings (excluding the target domain) on the PACS dataset.

Appendix C. Additional Experimental Results

C.1. Full results for overall comparison

C.2. Full results for ablation study

C.3. Sensitivity tests of varying loss weighting α and initial λ

Given (8), the loss weighting α basically determines how strongly the adversarial constraint should be applied to obtain a desirable mixup distribution. If α is too large, the training of the weight generation policy will collapse and tend to generate a distribution that closely mimics the source distribution. If α is too small, the constraint becomes ineffective, leading to unregularized extrapolation that deviates substantially from the reasonable region, thereby severely affecting base-model learning. Since the adversarial constraint prevents the detrimental over-extrapolation effect, the intensity of the training signal is related to the initialization of the scaling factor λ, which generally determines the extent of extrapolation.

Table C.8 presents the sensitivity tests of varying α and initial λ on model performance on the `TerraInc` dataset. We vary α in {0.01, 0.03, 0.1, 0.3} and initial λ in {1, 3, 5, 8}. Firstly, we observe that when the initial λ is set to 1 (no extrapolation in the beginning), a smaller α value results in better performance, as the lightly imposed adversarial constraint allows λ to learn useful extrapolation from the flatness-aware loss more freely. When we increase the initial λ to more than 1 (extrapolation exists from the beginning), we can see that a greater α is more beneficial, as the stronger adversarial constraint helps adjust mixup data that significantly drift from the source domains, which are detrimental for base model training. Overall, the model's performance is sensitive to λ. A large initial λ may simply crash the base model training and results in poor performance. This also underscores the importance of applying the adversarial constraint, which draws the extrapolated data closer to the source domains, thereby mitigating the adverse effects of large λ. In this specific case, we found that a moderately low value for α (around 0.1 to 0.3) yields the best result when the initial λ is 3.

C.4. Effect of mixup on the learned feature representations

Though FGMix performs mixup after the feature extractor g_θ to regularize the classifier, the gradients induced by mixup naturally propagate back into g_θ and can indirectly influence the learned features through end-to-end training.

To empirically examine the effect of mixup on the learned feature representations of feature extractor g_θ, we visualize the latent distributions produced by g_θ under FGMix (with mixup) and ERM (without mixup) on the PACS dataset using t-SNE, as shown in Fig. C.1. From the visualizations, several observations can be made: both ERM and FGMix learn well-separated class clusters,

Table C.1
Overall comparison of selected algorithms on `PACS`.

	Algorithm	A	C	P	S	Avg.
	ERM [51]	84.7±0.4	80.8±0.6	97.2±0.3	79.3±1.0	85.5
Mixup-based Methods	Mixup [12]	86.1±0.5	78.9±0.8	**97.6**±0.1	75.8±1.8	84.6
	Manifold Mixup† [9]	**88.8**±0.6	80.9±0.9	95.8±0.7	79.8±0.1	86.2
	MixStyle† [11]	83.7±0.1	80.7±3.7	95.5±1.2	**81.4**±2.6	85.3
	MetaMixup† [10]	84.9±1.5	79.6±0.6	96.3±0.9	80.1±1.4	85.2
Gradient-based Methods	PCGrad† [31]	85.9±1.0	80.4±0.1	95.5±0.1	78.2±2.2	85.0
	AND-Mask [13]	85.3±1.4	79.2±2.0	96.9±0.4	76.2±1.4	84.4
	Fish [15]	-	-	-	-	85.5
	Fishr [33]	88.4±0.2	78.7±0.7	97.0±0.1	77.8±2.0	85.5
Augmentation Methods	L2A-OT† [52]	87.9±1.5	81.4±2.3	96.7±1.3	77.2±2.4	85.8
	CNSN† [53]	86.7±0.2	80.2±1.1	96.2±0.7	79.2±1.5	85.6
	DDG† [54]	87.4±1.2	79.2±1.8	97.4±1.2	77.3±2.1	85.3
DomainBed SOTA	SelfReg [56]	87.9±1.0	79.4±1.4	96.8±0.7	78.3±1.2	85.6
	SagNet [55]	87.4±1.0	80.7±0.6	97.1±0.1	80.0±0.4	86.3
	CORAL [57]	88.3±0.2	80.0±0.5	97.5±0.3	78.8±1.3	86.2
	FGMix (ours)	87.2±1.3	**81.8**±1.2	97.3±0.7	80.7±1.6	**86.8**
Combined with flatness-aware solver SWA [21,37]						
	ERM + SWA†	88.1±0.6	82.0±0.6	96.9±0.5	80.8±0.2	87.0
	SelfReg + SWA	85.9±0.6	81.9±0.4	96.8±0.1	81.4±0.6	86.5
	CORAL + SWA†	87.6±0.3	83.1±0.7	**97.5**±0.2	81.6±0.8	87.5
	FGMix + SWA (ours)	**89.6**±0.7	**83.8**±0.5	97.2±0.5	**83.0**±0.8	**88.4**

Table C.2
Overall comparison of selected algorithms on VLCS.

	Algorithm	C	L	S	V	Avg.
	ERM [51]	97.7±0.4	64.3±0.9	73.4±0.5	74.6±1.3	77.5
Mixup-based Methods	Mixup [12]	98.3±0.6	64.8±1.0	72.1±0.5	74.3±0.8	77.4
	Manifold Mixup† [9]	97.4±1.3	63.8±1.9	**73.7**±0.0	72.0±1.3	76.7
	MixStyle† [11]	97.9±0.7	63.3±1.7	70.7±0.2	**77.5**±0.4	77.4
	MetaMixup† [10]	98.6±1.2	63.4±0.8	72.8±0.1	76.7±1.1	77.9
Gradient-based Methods	PCGrad† [31]	98.1±0.1	**66.2**±0.1	70.1±1.1	75.7±1.9	77.5
	AND-Mask [13]	97.8±0.4	64.3±1.2	73.5±0.7	76.8±2.6	78.1
	Fish [15]	-	-	-	-	77.8
	Fishr [33]	**98.9**±0.3	64.0±0.5	71.5±0.2	76.8±0.7	77.8
Augmentation Methods	L2A-OT† [52]	98.2±0.8	64.1±0.4	71.6±1.5	75.5±0.8	77.4
	CNSN† [53]	97.9±0.4	65.2±1.3	69.8±1.4	75.6±0.7	77.1
	DDG† [54]	98.5±0.9	64.2±1.7	70.2±0.8	74.3±1.4	76.8
DomainBed SOTA	SelfReg [56]	96.7±0.4	65.2±1.2	73.1±1.3	76.2±0.7	77.8
	SagNet [55]	97.9±0.4	64.5±0.5	71.4±1.3	**77.5**±0.5	77.8
	CORAL [57]	98.3±0.1	66.1±1.2	73.4±0.3	**77.5**±1.2	**78.8**
	FGMix (ours)	98.1±0.4	63.9±0.9	73.1±1.4	77.1±0.8	78.1
Combined with flatness-aware solver SWA [21,37]						
	ERM + SWA†	97.0±0.9	**64.3**±0.8	72.4±0.6	75.2±0.1	77.2
	SelfReg + SWA	97.4±0.4	63.5±0.3	72.6±0.1	76.7±0.7	77.5
	CORAL + SWA†	**98.6**±0.3	63.2±0.2	72.8±0.2	**78.2**±1.1	78.2
	FGMix + SWA (ours)	**98.6**±0.4	63.5±0.1	**74.7**±0.6	78.1±1.2	**78.7**

Table C.3
Overall comparison of selected algorithms on OfficeHome.

	Algorithm	A	C	P	R	Avg.
	ERM [51]	61.3±0.7	52.4±0.3	75.8±0.1	76.6±0.3	66.5
Mixup-based Methods	Mixup [12]	62.4±0.8	54.8±0.6	**76.9**±0.3	78.3±0.2	68.1
	Manifold Mixup† [9]	61.6±0.8	**55.1**±3.1	75.6±0.2	77.0±0.0	67.3
	MixStyle† [11]	61.7±0.6	53.3±1.6	76.3±1.0	77.8±0.5	67.3
	MetaMixup† [10]	63.5±1.1	54.6±0.4	75.9±0.9	78.7±0.4	68.2
Gradient-based Methods	PCGrad† [31]	60.6±1.1	52.1±1.7	74.4±0.6	74.0±0.3	65.5
	AND-Mask [13]	59.5±1.2	51.7±0.2	73.9±0.4	77.1±0.2	65.6
	Fish [15]	-	-	-	-	68.6
	Fishr [33]	62.4±0.5	54.4±0.4	76.2±0.5	78.3±0.1	67.8
Augmentation Methods	L2A-OT† [52]	64.5±1.0	53.8±2.8	76.2±1.3	77.9±1.1	68.1
	CNSN† [53]	63.1±0.7	53.2±2.1	74.1±1.3	78.6±0.6	67.3
	DDG† [54]	63.7±0.8	54.5±1.2	75.9±1.0	78.2±0.9	68.1
DomainBed SOTA	SelfReg [56]	63.6±1.4	53.1±1.0	**76.9**±0.4	78.1±0.4	67.9
	SagNet [55]	63.4±0.2	54.8±0.4	75.8±0.4	78.3±0.3	68.1
	CORAL [57]	**65.3**±0.4	54.4±0.5	76.5±0.1	78.4±0.5	68.7
	FGMix (ours)	65.2±1.5	54.6±2.1	**76.9**±0.8	**79.2**±0.8	**69.0**
Combined with flatness-aware solver SWA [21,37]						
	ERM + SWA†	65.7±1.1	56.2±0.4	77.3±0.1	78.9±0.6	69.5
	SelfReg + SWA	64.9±0.8	55.4±0.6	78.4±0.2	78.8±0.1	69.4
	CORAL + SWA†	68.3±0.2	57.0±0.0	77.9±0.3	79.7±0.0	70.7
	FGMix + SWA (ours)	**68.5**±1.0	**57.6**±0.8	**78.8**±0.5	**80.4**±0.2	**71.3**

Table C.4
Overall comparison of selected algorithms on `TerraIncognita`.

	Algorithm	L100	L38	L43	L46	Avg.
	ERM [51]	49.8±4.4	42.1±1.4	56.9±1.8	35.7±3.9	46.1
Mixup-based Methods	Mixup [12]	**59.6**±2.0	42.2±1.4	55.9±0.8	33.9±1.4	47.9
	Manifold Mixup† [9]	57.7±2.9	42.4±3.0	55.8±0.4	39.2±1.9	48.8
	MixStyle† [11]	53.9±0.6	42.3±0.9	53.2±2.7	37.6±0.1	46.8
	MetaMixup† [10]	53.2±1.7	43.7±2.6	54.5±1.9	36.9±1.1	47.1
Gradient-based Methods	PCGrad† [31]	55.9±2.0	43.2±3.8	56.4±2.5	37.7±1.0	48.3
	AND-Mask [13]	50.0±2.9	40.2±0.8	53.3±0.7	34.8±1.9	44.6
	Fish [15]	-	-	-	-	45.1
	Fishr [33]	50.2±3.9	43.9±0.8	55.7±2.2	39.8±1.0	47.4
Augmentation Methods	L2A-OT† [52]	58.7±2.0	43.5±2.7	54.8±1.9	37.4±1.8	48.6
	CNSN† [53]	58.2±1.9	43.2±2.1	57.1±1.3	35.2±1.1	48.4
	DDG† [54]	**59.6**±3.0	41.2±2.4	56.0±1.8	33.9±1.2	47.7
DomainBed SOTA	SelfReg [56]	48.8±0.9	41.3±1.8	57.3±0.7	40.6±0.9	47.0
	SagNet [55]	53.0±2.9	43.0±2.5	**57.9**±0.6	40.4±1.3	48.6
	CORAL [57]	51.6±2.4	42.2±1.0	57.0±1.0	39.8±2.9	47.6
	FGMix (ours)	54.1±2.7	**44.7**±1.1	57.0±1.8	**40.9**±0.8	**49.2**
Combined with flatness-aware solver SWA [21,37]						
	ERM + SWA†	53.5±0.9	47.6±0.8	58.2±0.4	41.0±0.8	50.1
	SelfReg + SWA	56.8±0.9	44.7±0.6	59.6±0.3	**42.9**±0.8	51.0
	CORAL + SWA†	55.6±0.5	48.1±0.7	58.5±0.1	42.2±1.0	51.1
	FGMix + SWA (ours)	**57.1**±0.8	**49.5**±1.3	**60.7**±0.3	41.8±1.1	**52.3**

Table C.5
Overall comparison of selected algorithms on `DomainNet`.

	Algorithm	clip	info	paint	quick	real	sketch	Avg.
	ERM [51]	58.1±0.3	18.8±0.3	46.7±0.3	12.2±0.4	59.6±0.1	49.8±0.4	40.9
Mixup-based Methods	Mixup [12]	55.7±0.3	18.5±0.5	44.3±0.5	12.5±0.4	55.8±0.3	48.2±0.5	39.2
	Manifold Mixup† [9]	60.7±0.4	19.4±0.1	47.1±0.3	11.4±0.2	59.6±0.6	48.7±0.2	41.2
	MixStyle† [11]	59.9±0.2	19.0±0.3	47.0±0.1	11.5±0.1	58.9±0.4	48.8±0.0	40.9
	MetaMixup† [10]	60.7±0.3	20.0±0.5	47.1±0.4	12.8±0.1	60.1±0.1	50.1±0.3	41.8
Gradient-based Methods	PCGrad† [31]	60.3±0.2	18.1±0.4	47.0±0.4	12.9±0.1	59.8±0.0	48.4±0.3	41.1
	AND-Mask [13]	52.3±0.8	16.6±0.3	41.6±1.1	11.3±0.1	55.8±0.4	45.4±0.9	37.2
	Fish [15]	-	-	-	-	-	-	**42.7**
	Fishr [33]	58.2±0.5	20.2±0.2	**47.7**±0.3	12.7±0.2	60.3±0.2	**50.8**±0.1	41.7
Augmentation Methods	L2A-OT† [52]	58.7±0.3	18.5±0.4	46.2±0.5	11.3±0.7	57.6±0.8	48.9±0.2	40.2
	CNSN† [53]	60.2±0.2	19.0±0.1	46.3±0.5	11.6±0.3	58.2±0.6	48.7±0.4	40.7
	DDG† [54]	59.7±0.5	18.7±0.2	45.1±0.9	12.1±0.4	56.8±0.3	47.5±0.7	40.0
DomainBed SOTA	SelfReg [56]	58.5±0.1	**20.7**±0.1	47.3±0.3	13.1±0.3	58.2±0.2	51.1±0.3	41.5
	SagNet [55]	57.7±0.3	19.0±0.2	45.3±0.3	12.7±0.5	58.1±0.5	48.8±0.2	40.3
	CORAL [57]	59.2±0.1	19.7±0.2	46.6±0.3	**13.4**±0.4	59.8±0.2	50.1±0.6	41.5
	FGMix (ours)	**61.1**±0.1	20.4±0.6	47.0±0.6	12.5±0.5	**61.1**±0.8	50.7±0.3	42.1
Combined with flatness-aware solver SWA [21,37]								
	ERM + SWA†	62.0±0.1	21.0±0.0	50.8±0.2	13.6±0.3	62.7±0.1	**54.0**±0.2	44.0
	SelfReg + SWA	62.4±0.1	**22.6**±0.1	51.8±0.1	14.3±0.1	62.5±0.2	53.8±0.3	44.6
	CORAL + SWA†	63.0±0.3	22.1±0.5	51.7±0.4	14.9±0.5	63.0±0.3	53.1±0.1	44.6
	FGMix + SWA (ours)	**63.1**±0.2	22.3±0.7	**52.2**±0.5	**15.1**±0.2	**63.9**±0.2	**54.0**±0.3	**45.1**

Table C.6
Ablation study of FGMix on PACS.

Variant	similarity-based weights	gradient-based similarity	scaling & shifting	$\mathcal{L}_{flat}$	$\mathcal{L}_{adv}$	A	C	P	S	Avg.
A (baseline)						84.3±0.8	80.2±0.7	94.8±0.3	79.6±1.0	84.7
B	✓					85.1±0.4	80.9±0.8	95.3±0.2	79.6±0.6	85.2
C	✓	✓				86.6±0.2	80.1±0.5	95.9±0.1	79.7±0.1	85.6
D	✓	✓	✓			87.5±1.5	81.0±1.4	95.6±0.7	80.0±0.7	86.0
E	✓	✓	✓	✓		**88.0**±1.7	81.5±2.3	96.9±1.4	80.3±1.4	86.7
F	✓		✓	✓	✓	87.9±1.2	80.9±1.9	96.1±1.3	80.0±1.6	86.2
FGMix (ours)	✓	✓	✓	✓	✓	87.2±1.3	**81.8**±1.2	**97.3**±0.7	**80.7**±1.6	**86.8**

Table C.7
Ablation study of FGMix on TerraIncognita.

Variant	similarity-based weights	gradient-based similarity	scaling & shifting	$\mathcal{L}_{flat}$	$\mathcal{L}_{adv}$	L100	L38	L43	L46	Avg.
A (baseline)						51.1±0.7	43.2±1.4	56.4±1.2	39.3±1.1	47.5
B	✓					50.4±1.0	43.9±1.2	57.0±0.8	40.1±0.5	47.9
C	✓	✓				50.4±0.8	**44.7**±1.6	57.5±0.6	39.9±1.3	48.1
D	✓	✓	✓			51.9±1.7	44.2±2.0	58.0±1.1	**41.1**±1.9	48.8
E	✓	✓	✓	✓		54.0±2.1	44.3±3.0	**58.1**±1.9	40.5±1.8	**49.2**
F	✓		✓	✓	✓	53.1±1.9	44.0±2.2	57.9±1.2	40.7±1.6	48.9
FGMix (ours)	✓	✓	✓	✓	✓	**54.1**±1.7	**44.7**±1.1	57.0±1.8	40.9±0.8	**49.2**

Table C.8
Sensitivity tests of varying loss weighting α and initial λ on TerraInc dataset.

	α for $\mathcal{L}_{adv}$			
initial λ	0.01	0.03	0.1	0.3
1	47.9	48.2	46.3	46.5
3	47.3	47.1	**49.2**	49.0
5	44.3	44.0	46.5	46.1
8	43.1	44.0	44.8	45.1

demonstrating that both methods produce a strong class-discriminative structure of the latent space; ERM exhibits more pronounced domain-specific subclusters within each class (e.g., “elephant” and “dog”), where samples from the Cartoon domain (orange) form a subcluster that is visibly segregated from the other source domains; in contrast, FGMix produces more cohesive, domain-mixed clusters in which samples from different domains are more evenly distributed within each class cluster, and the domain-specific structure is noticeably reduced. A possible explanation is that mixing latent codes from different domains within the same class encourages the classifier to rely less on domain-specific artifacts. As a result, the induced gradients promote the learning of features that emphasize class-relevant information while deemphasizing domain-induced variations. This induced feature geometry may also facilitate the robust classification of extrapolated out-of-domain representations under parameter perturbations, and we leave a more thorough investigation of this effect to future work.

While this improvement in latent-space cohesion is a byproduct of mixup training, the main benefit of FGMix still lies in its ability to extrapolate beyond the convex hull of the source domains-an effect neither ERM nor interpolation-only mixup achieves (as shown previously in Fig. 4). This is also reflected in the t-SNE plots, where target-domain samples (Sketch, pink) remain entangled near the center along class boundaries, highlighting where extrapolated mixup instances are particularly helpful.

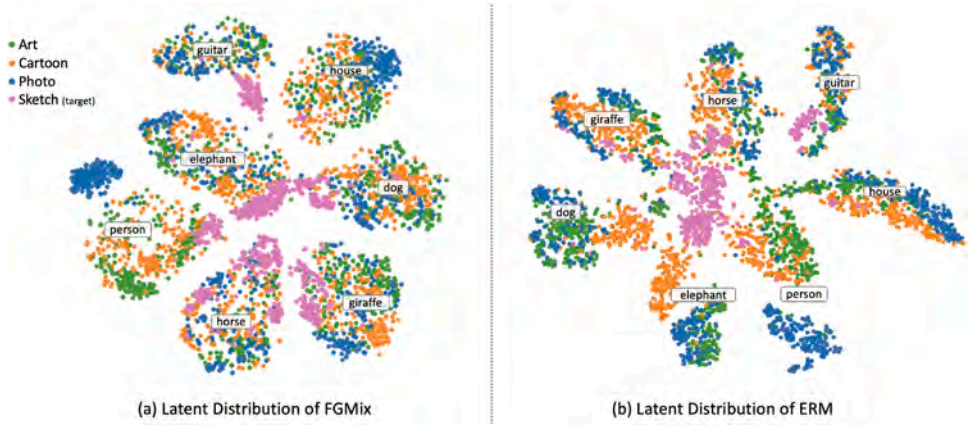


Fig. C.1. t-SNE visualization of the latent distributions on `PACS` trained with (a) FGMix and (b) ERM. In both plots, we use "`Art`", "`Cartoon`", "`Photo`" as the source domains and "`Sketch`" as the target domain.

Appendix D. 2D Loss/Accuracy Surface Construction

Following Izmailov et al. [21], Cha et al. [36], we select 3 model weights $\{w_1, w_2, w_3\}$ at the initialization, the end of ERM + SWA optimization path, and the end of FGMix + SWA optimization path, respectively. Note that w_i is the concatenation of vectorized θ_i and ϕ_i for the ith selected model. We define the orthogonal basis $\{u, v\}$ of the 2D plane as:

$$u = w_2 - w_1, \quad v = \frac{(w_3 - w_1) - \langle w_3 - w_1, w_2 - w_1 \rangle}{\|w_2 - w_1\|^2 \cdot (w_2 - w_1)}.$$

The orthonormal basis of the 2D plane is $\hat{u} = u/\|u\|$ and $\hat{v} = v/\|v\|$.

We project the optimization path onto the 2D plane by computing the coordinates of each point on the path. That is, for the point at the jth step, its u- and v-coordinates are $\langle (w_j - w_1), \hat{u} \rangle$ and $\langle (w_j - w_1), \hat{v} \rangle$ respectively. To plot the loss surface, we define ranges on u- and v-axes that fully contain the optimization paths, say $[u_1, u_2]$ for u-axis and $[v_1, v_2]$ for v-axis. We then obtain the model weight corresponding to each grid point in the defined range, i.e., $w = w_1 + a\hat{u} + b\hat{v}$, $\forall a \in [u_1, u_2], b \in [v_1, v_2]$, and compute the loss/accuracy of the entire training/test dataset using the model weight. The loss/accuracy values are visualized with a contour plot. For direct comparison, we use the same u, v ranges for all three plots (ERM & FGMix train loss plots and test accuracy plot), and set the scale interval to be the same for the two train loss plots for flatness comparison.

References

[1] G. Blanchard, G. Lee, C. Scott, Generalizing from several related classification tasks to a new unlabeled sample, Adv. Neural Inf. Process. Syst. 24 (2011) 2178–2186.
[2] K. Muandet, D. Balduzzi, B. Schölkopf, Domain generalization via invariant feature representation, in: International Conference on Machine Learning, PMLR, 2013, pp. 10–18.
[3] H. Li, S.J. Pan, S. Wang, A.C. Kot, Domain generalization with adversarial feature learning, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 5400–5409.
[4] D. Li, Y. Yang, Y.-Z. Song, T.M. Hospedales, Deeper, broader and artier domain generalization, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 5542–5550.
[5] D. Li, Y. Yang, Y.-Z. Song, T.M. Hospedales, Learning to generalize: meta-learning for domain generalization, in: Thirty-Second AAAI Conference on Artificial Intelligence, 2018.
[6] Y. Balaji, S. Sankaranarayanan, R. Chellappa, MetaReg: towards domain generalization using meta-regularization, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018, pp. 1006–1016.
[7] D. Li, J. Zhang, Y. Yang, C. Liu, Y.-Z. Song, T.M. Hospedales, Episodic training for domain generalization, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1446–1455.
[8] H. Zhang, M. Cisse, Y.N. Dauphin, D. Lopez-Paz, Mixup: beyond empirical risk minimization, in: International Conference on Learning Representations, 2018.
[9] V. Verma, A. Lamb, C. Beckham, A. Najafi, I. Mitliagkas, D. Lopez-Paz, Y. Bengio, Manifold mixup: better representations by interpolating hidden states, in: International Conference on Machine Learning, PMLR, 2019, pp. 6438–6447.
[10] Z. Mai, G. Hu, D. Chen, F. Shen, H.T. Shen, MetaMixUp: learning adaptive interpolation policy of MixUp with metalearning, IEEE Trans. Neural Netw. Learn. Syst. 33 (2021) 3050–3064.
[11] K. Zhou, Y. Yang, Y. Qiao, T. Xiang, Domain generalization with mixstyle, in: International Conference on Learning Representations, 2020.
[12] Y. Wang, H. Li, A.C. Kot, Heterogeneous domain generalization via domain mixup, in: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2020, pp. 3622–3626.
[13] G. Parascandolo, A. Neitz, A. ORVIETO, L. Gresele, B. Schölkopf, Learning explanations that are hard to vary, in: International Conference on Learning Representations, 2020.
[14] L. Mansilla, R. Echeveste, D.H. Milone, E. Ferrante, Domain generalization via gradient surgery, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 6630–6638.
[15] Y. Shi, J. Seely, P.H.S. Torr, N. Siddharth, A. Hannun, N. Usunier, G. Synnaeve, Gradient Matching for Domain Generalization, (2021). arXiv preprint arXiv:2104.09937.
[16] A.E. Eshratifar, D. Eigen, M. Pedram, Gradient agreement as an optimization objective for meta-learning, (2018). arXiv preprint arXiv:1810.08178.

[17] Y. Du, W.M. Czarnecki, S.M. Jayakumar, M. Farajtabar, R. Pascanu, B. Lakshminarayanan, Adapting auxiliary losses using gradient similarity, (2018). arXiv preprint arXiv:1812.02224.
[18] S. Hochreiter, J. urgen Schmidhuber, C. Elvezia, Flat minima, Neural Comput. 9 (1) (1997) 1–42.
[19] P. Chaudhari, A. Choromanska, S. Soatto, Y. LeCun, C. Baldassi, C. Borgs, J. Chayes, L. Sagun, R. Zecchina, Entropy-SGD: biasing gradient descent into wide valleys (international conference on learning representations, ICLR 2017), in: 5th International Conference on Learning Representations, ICLR 2017, 2019.
[20] P. Foret, A. Kleiner, H. Mobahi, B. Neyshabur, Sharpness-aware minimization for efficiently improving generalization, in: International Conference on Learning Representations, 2020.
[21] P. Izmailov, A.G. Wilson, D. Podoprikhin, D. Vetrov, T. Garipov, Averaging weights leads to wider optima and better generalization, in: 34th Conference on Uncertainty in Artificial Intelligence 2018, UAI 2018, 2018, pp. 876–885.
[22] H. Guo, J. Jin, B. Liu, Stochastic Weight Averaging Revisited, (2022). arXiv preprint arXiv:2201.00519.
[23] O. Chapelle, J. Weston, L. Bottou, V. Vapnik, Vicinal risk minimization, in: NIPS, 2000.
[24] S. Yun, D. Han, S.J. Oh, S. Chun, J. Choe, Y. Yoo, CutMix: regularization strategy to train strong classifiers with localizable features, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 6023–6032.
[25] H.-P. Chou, S.-C. Chang, J.-Y. Pan, W. Wei, D.-C. Juan, Remix: rebalanced mixup, in: European Conference on Computer Vision, Springer, 2020, pp. 95–110.
[26] G. Giannone, S. Havrylov, J. Massiah, E. Yilmaz, Y. Jiao, Just Mix Once: Worst-group Generalization by Group Interpolation, (2022). arXiv preprint arXiv:2210.12195.
[27] I. Hwang, S. Lee, Y. Kwak, S.J. Oh, D. Teney, J.-H. Kim, B.-T. Zhang, Selecmix: debiased learning by contradicting-pair sampling, Adv. Neural Inf. Process. Syst. 35 (2022) 14345–14357.
[28] H. Yao, Y. Wang, S. Li, L. Zhang, W. Liang, J. Zou, C. Finn, Improving out-of-distribution robustness via selective augmentation, in: International Conference on Machine Learning, PMLR, 2022, pp. 25407–25437.
[29] H. Guo, Y. Mao, R. Zhang, Mixup as locally linear out-of-manifold regularization, in: Proceedings of the AAAI Conference on Artificial Intelligence, 33, 2019, pp. 3714–3722.
[30] M. Riemer, I. Cases, R. Ajemian, M. Liu, I. Rish, Y. Tu, G. Tesauro, Learning to learn without forgetting by maximizing transfer and minimizing interference, in: International Conference on Learning Representations, 2018.
[31] T. Yu, S. Kumar, A. Gupta, S. Levine, K. Hausman, C. Finn, Gradient surgery for multi-task learning, Adv. Neural Inf. Process. Syst. 33 (2020) 5824–5836.
[32] S. Shahtalebi, J.-C. Gagnon-Audet, T. Laleh, M. Faramarzi, K. Ahuja, I. Rish, Sand-mask: An enhanced gradient masking strategy for the discovery of invariances in domain generalization, (2021). arXiv preprint arXiv:2106.02266.
[33] A. Rame, C. Dancette, M. Cord, Fishr: Invariant gradient variances for out-of-distribution generalization, (2021). arXiv preprint arXiv:2109.02934.
[34] N.S. Keskar, J. Nocedal, P.T.P. Tang, D. Mudigere, M. Smelyanskiy, On large-batch training for deep learning: generalization gap and sharp minima, in: 5th International Conference on Learning Representations, ICLR 2017, 2017.
[35] D.J.C. MacKay, A practical Bayesian framework for backpropagation networks, Neural Comput. 4 (3) (1992) 448–472.
[36] J. Cha, S. Chun, K. Lee, H.-C. Cho, S. Park, Y. Lee, S. Park, SWAD: Domain generalization by seeking flat minima, in: Advances in Neural Information Processing Systems, 2021.
[37] D. Arpit, H. Wang, Y. Zhou, C. Xiong, Ensemble of averages: Improving model selection and boosting performance in domain generalization, (2021). arXiv preprint arXiv:2110.10832.
[38] D. Berthelot, C. Raffel, A. Roy, I. Goodfellow, Understanding and improving interpolation in autoencoders via an adversarial regularizer, in: International Conference on Learning Representations, 2018.
[39] F. Sung, Y. Yang, L. Zhang, T. Xiang, P.H.S. Torr, T.M. Hospedales, Learning to compare: relation network for few-shot learning, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 1199–1208.
[40] X. He, L. Liao, H. Zhang, L. Nie, X. Hu, T.-S. Chua, Neural collaborative filtering, in: Proceedings of the 26th International Conference on World Wide Web, 2017, pp. 173–182.
[41] H. Petzka, M. Kamp, L. Adilova, C. Sminchisescu, M. Boley, Relative flatness and generalization, in: Advances in Neural Information Processing Systems, 2021.
[42] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, Y. Bengio, Generative adversarial nets, Adv. Neural Inf. Process. Syst. 27 (2014) 2672–2680.
[43] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, B. Frey, Adversarial autoencoders, (2015). arXiv preprint arXiv:1511.05644.
[44] I. Gulrajani, D. Lopez-Paz, In search of lost domain generalization, (2020). arXiv preprint arXiv:2007.01434.
[45] C. Fang, Y. Xu, D.N. Rockmore, Unbiased metric learning: on the utilization of multiple datasets and web images for softening bias, in: Proceedings of the IEEE International Conference on Computer Vision, 2013, pp. 1657–1664.
[46] H. Venkateswara, J. Eusebio, S. Chakraborty, S. Panchanathan, Deep hashing network for unsupervised domain adaptation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 5018–5027.
[47] S. Beery, G. Van Horn, P. Perona, Recognition in terra incognita, in: Proceedings of the European Conference on Computer Vision (ECCV), 2018, pp. 456–473.
[48] X. Peng, Q. Bai, X. Xia, Z. Huang, K. Saenko, B. Wang, Moment matching for multi-source domain adaptation, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 1406–1415.
[49] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
[50] P. Teterwak, K. Saito, T. Tsiligkaridis, K. Saenko, B.A. Plummer, ERM++: An Improved Baseline for Domain Generalization, (2023). arXiv preprint arXiv:2304.01973.
[51] V.N. Vapnick, Statistical Learning Theory, Wiley, New York, 1998.
[52] K. Zhou, Y. Yang, T. Hospedales, T. Xiang, Learning to generate novel domains for domain generalization, in: European Conference on Computer Vision, Springer, 2020, pp. 561–578.
[53] Z. Tang, Y. Gao, Y. Zhu, Z. Zhang, M. Li, D.N. Metaxas, Crossnorm and selfnorm for generalization under distribution shifts, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 52–61.
[54] H. Zhang, Y.-F. Zhang, W. Liu, A. Weller, B. Schölkopf, E.P. Xing, Towards principled disentanglement for domain generalization, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2022, pp. 8024–8034.
[55] H. Nam, H. Lee, J. Park, W. Yoon, D. Yoo, Reducing domain gap by reducing style bias, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021, pp. 8690–8699.
[56] D. Kim, Y. Yoo, S. Park, J. Kim, J. Lee, SelfReg: self-supervised contrastive regularization for domain generalization, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2021, pp. 9619–9628.
[57] B. Sun, K. Saenko, Deep coral: correlation alignment for deep domain adaptation, in: European Conference on Computer Vision, Springer, 2016, pp. 443–450.