A OVERVIEW

In the appendix we give the full breakdown of results as well as additional results in appendix B. We then give a thorough literature review in appendix C; a description of the datasets and how we set up the distribution shifts and conditions on these datasets in appendix D; code samples from our framework showing how it can be extended in appendix E; further details about each method evaluated in appendix F; and finally implementation details in appendix G.

B RESULTS

We give the complete breakdown of results for all methods and shifts in appendix B.1. We give a detailed analysis on the impact of each augmentation type in RandAugment in appendix B.2. Finally, we evaluate the performance using a different attribute as the label in appendix B.3 and using the ID (as opposed to OOD) validation set on iWildCam and Camelyon17 in appendix B.4.

B.1 COMPLETE RESULTS

Here, we give the complete results over each dataset for each shift (*spurious correlation*, *low-data drift*, and *unseen data shift*) in figures 10-12. In these plots, we plot the mean and standard deviation of each method on each dataset and sort them according to their performance. The bars for each method are colored according to the general method they belong to (green denotes different architectures, orange different heuristic augmentation methods, and so on).

B.2 A DETAILED ANALYSIS ON THE IMPACT OF AUGMENTATION

We found that different methods of performing heuristic augmentation perform differently: some outperform the ERM baseline, some do not. Each of these methods is composed of individual augmentation techniques. Here we investigate how each technique contributes to the end performance and thereby how the *choice* of augmentation function impacts the robustness of the models.

We take the augmentations used in RandAugment. Instead of sampling all augmentations randomly, for each augmentation, we randomly sample the given augmentation or the identity function. We plot the deviation from the mean in figure 8 for each dataset under the setting *unseen data shift*. The results are surprising. No augmentation always leads to a strong boost in performance. For example, using *invert* improves performance on dSprites, Shapes3D but harms performance on MPI3D, SmallNorb, and iWildCam. Similarly, using *color* improves performance on most datasets but harms performance on Camelyon17.

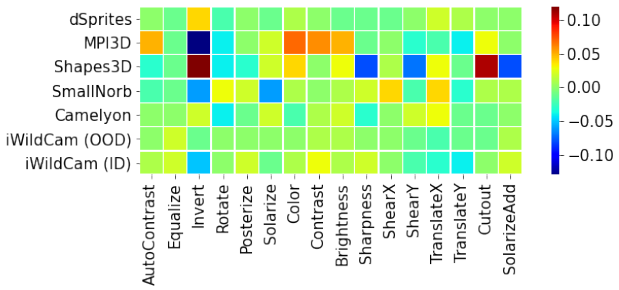


Figure 8: **RandAugment ablation.** Performance of each augmentation across the different datasets. As can be seen, no one augmentation always improves performance. An augmentation that may improve performance on one dataset (e.g. invert on Shapes3D) hurts performance on another (e.g. invert on SmallNorb).

B.3 RESULTS WITH DIFFERENT ATTRIBUTES

Here we investigate if the results are dependent on the attributes being investigated. We investigate *unseen data shift* on dSprites, but instead of predicting shape, we predict color. We then leave out

some shapes and test whether models can correctly predict color given the unseen shape. We find that *all* methods generalise to the OOD case with approximately perfect score, as shown in figure 9.

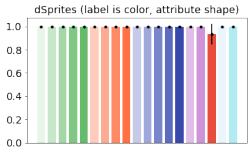
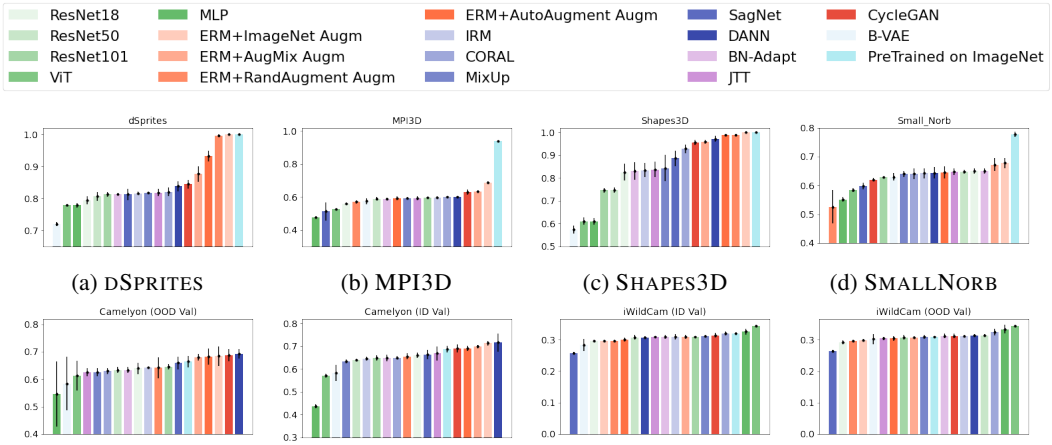


Figure 9: **Shift 3: Unseen data shift.** We plot the top-1 accuracy (y-axis) for the DSPRITES dataset. Unlike in the main paper, here the label is the color and the attribute the shape. Higher is better. All approaches solve the task.

B.4 RESULTS USING DIFFERENT, OOD VALIDATION SETS

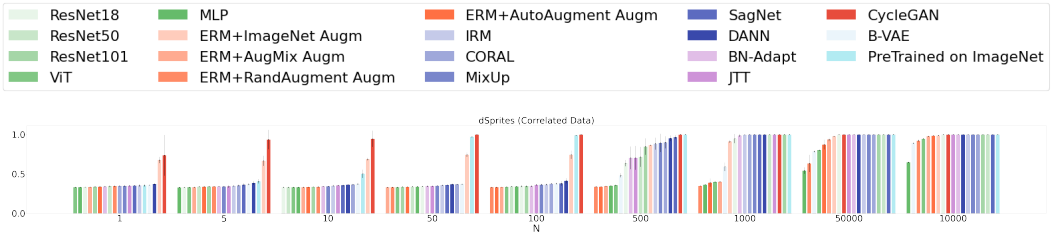
When evaluating on iWILDCAM and CAMELYON17, we explore whether using the out-of-distribution (OOD) or in-distribution (ID) validation sets is preferable for obtaining best performance. We find that neither the out-of-distribution (OOD) nor in-distribution (ID) validation sets perform best, but the relative performance remains similar. The relative performance of different models in figure 10 when using the ID or OOD validation set is comparable on CAMELYON17 and iWILDCAM. However, the maximum performance is not consistently better using either the ID or OOD set (for CAMELYON17, using the OOD set performs a bit better but for iWILDCAM, using the ID set performs best).



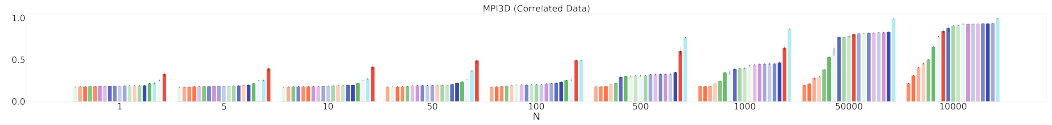
(a) DSPRITES (b) MPI3D (c) SHAPES3D (d) SMALLNORB

(e) CAMELYON17. Using the ID val set. (f) CAMELYON17. Using the OOD val set. (g) IWILDCAM. Using the ID val set. (h) IWILDCAM. Using the OOD val set.

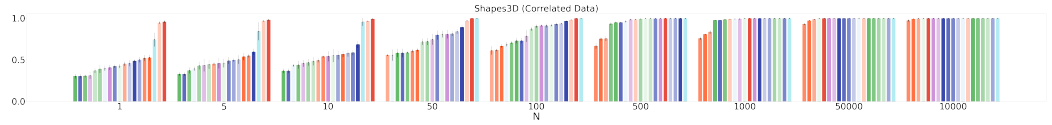
Figure 10: **Shift 3: Unseen data.** We plot the top-1 accuracy (y-axis) over different datasets. For each value of N we resort the results and show them in order. Higher is better.



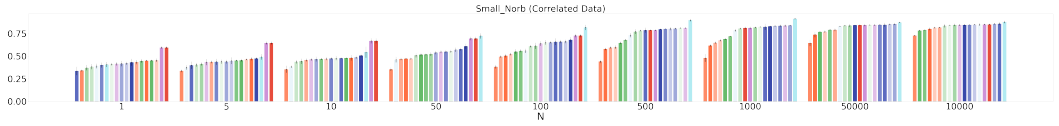
(a) DSPRITES



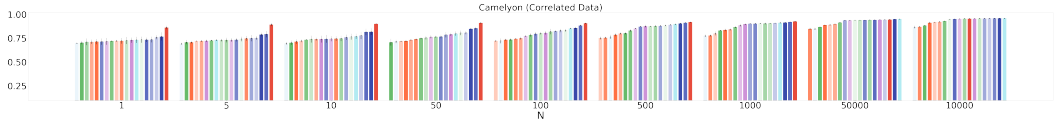
(b) MPI3D



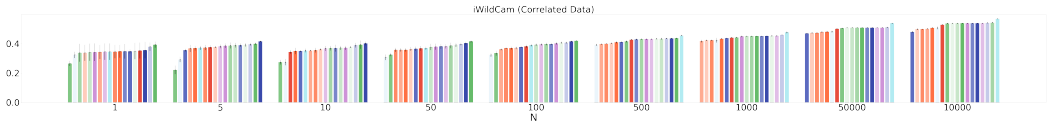
(c) SHAPES3D



(d) SMALLNORB

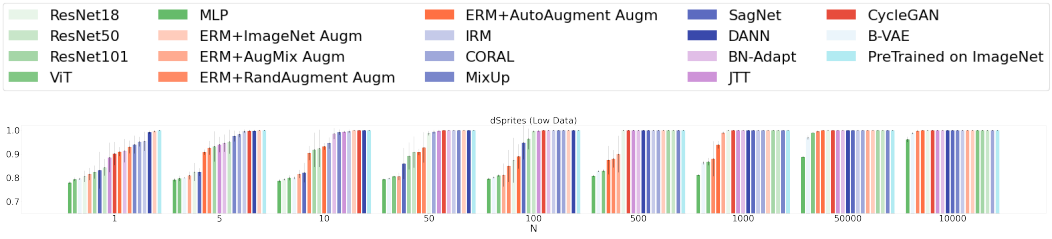


(e) CAMELYON17

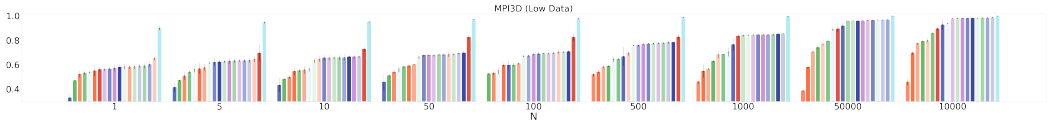


(f) IWILDCAM

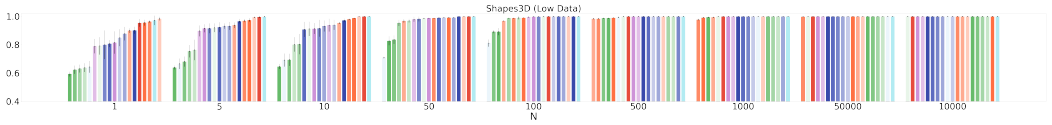
Figure 11: **Shift 1: Spurious correlation.** We plot the top-1 accuracy (y-axis) for different values of N (the number of samples from the independent distribution, the x-axis) over different datasets. For each value of N we resort the results and show them in order. Higher is better.



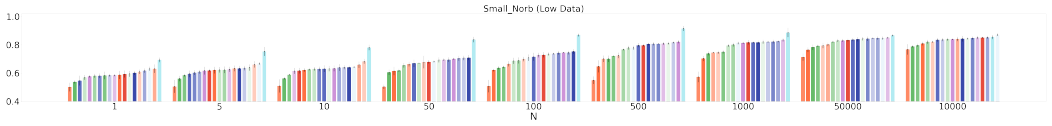
(a) DSPRITES



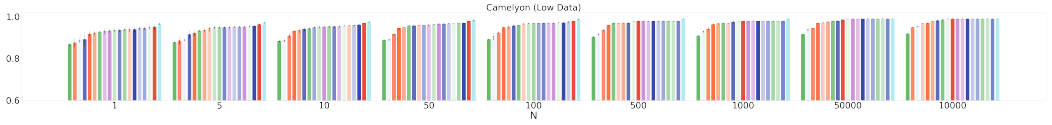
(b) MPI3D



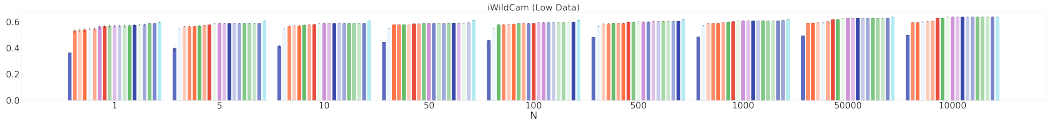
(c) SHAPES3D



(d) SMALLNORB



(e) CAMELYON17



(f) IWILDCAM

Figure 12: **Shift 2: Low data drift.** We plot the top-1 accuracy (y-axis) for different values of N (the number of samples from the independent distribution, the x-axis) over different datasets. For each value of N we resort the results and show them in order. Higher is better.

	Controllable shifts?	Many methods?	Real world motivated?
Ours	✓	✓	✓
WILDS (Koh et al., 2020)	✗	✗	✓
Gulrajani & Lopez-Paz (2021)	✗	✓	✗
Hendrycks et al. (2021)	✗	✓	✗

Table 1: Comparison of the three most similar benchmark works to ours. Unlike these works, our framework allows for controlling the distribution shift to be analysed. Moreover, we use real-world motivated datasets and evaluate methods across multiple approaches; we encompass more types of approaches than Gulrajani & Lopez-Paz (2021) and Hendrycks et al. (2021).

C LITERATURE REVIEW

Here we discuss in depth related literature. We first discuss datasets used to evaluate different distribution shifts. We then discuss methods related to the five common approaches we consider in the paper: architecture choice, data augmentation, domain generalization methods, adaptive approaches, and representation learning. We additionally discuss the hypotheses present in the current literature and whether our results corroborate or dispute those hypotheses.

Datasets to evaluate distribution shifts. Obtaining datasets of real world distribution shifts is challenging and expensive. As a result, many works have focussed on using synthetic or existing datasets to build their benchmarks. One approach is to create synthetic shifts over ImageNet to study how models generalise in specific, synthetic conditions: ImageNet-C (Hendrycks & Dietterich, 2019) studies the impact of standard corruptions; Stylized ImageNet (Geirhos et al., 2019) studies the impact of texture; Waterbirds and the Backgrounds challenge (Sagawa et al., 2020; Xiao et al., 2020) study the impact of a fake background; and colored MNIST (Gulrajani & Lopez-Paz, 2021) studies the impact of color on classification. Another approach is to consider how models trained on a given dataset generalise to other datasets of the same set of objects (e.g. cartoons to paintings in PACS or ImageNet to ImageNetV2) (Torralba & Efros, 2011; Li et al., 2017; Recht et al., 2019; Peng et al., 2019; Venkateswara et al., 2017) or over time (Shankar et al., 2019). While these datasets give insight into the biases of a trained model, models that do well on them may not necessarily do well on real world tasks. As a result, the WILDS dataset (Koh et al., 2020) was created as a collection of real world distribution shifts where the OOD task is well defined (e.g. the model should generalise to unseen countries for satellite imagery or hospitals for tumour identification) in order to evaluate progress.

Our framework is complementary to these datasets (table 1). Given a dataset, our framework can be used to set up a desired distribution shift to be investigated with fine-grained control over the type of distribution shift and amount of shift. Moreover, our framework provides extendable classes for a wide range of approaches and datasets. Finally, we present a robustness framework inspired by disentangling literature analyzing *when* we expect models to be able to generalise across these shifts.

Frameworks to evaluate the impact of label noise. A related area of work evaluates the impact of label noise on downstream performance. To ablate the impact of label noise, this was originally studied by constructing synthetic noise on standard datasets (Han et al., 2018; Hendrycks et al., 2018; Khetan et al., 2018; Patrini et al., 2017) in a variety of conditions: some labels are trusted (Hendrycks et al., 2018), there is a fixed label budget (Khetan et al., 2018), or there is knowledge of the confusion matrix (Patrini et al., 2017). Gu et al. (2021) generalised this framework to account for auxiliary information (such as rater expertise) and the varying difficulty of samples. Our framework mostly focuses on distribution shifts but can be extended to include more complex types of label noise.

Architecture choice. Previous work has investigated the impact of model size on robustness to various distribution shifts, finding that larger models generally are more robust when considering common corruptions (Hendrycks et al., 2019) and adversarial training (Xie & Yuille, 2020). However, (Schott et al., 2021) finds that larger models do not give representations that generalise better

within the same domain. We find that there is no strict rule correlating model size or model capacity with performance. Sometimes deeper ResNets perform best, sometimes not. The ViT model (with the highest capacity) often performs worse, but on iWildCam it performs best. However, we note that using additional data or pretraining on all models or methods may change their relative performance.

Data augmentation. Data augmentation is often pivotal to achieve state of the art performance on machine learning benchmarks, leading to a wide range of methods to perform heuristic data augmentation (He et al., 2016; Hendrycks et al., 2020; Cubuk et al., 2019; 2020; Zhang et al., 2018). When considering domain generalization, more specific techniques for augmentation have been devised to improve a model's ability to generalise using MixUp (Gulrajani & Lopez-Paz, 2021; Xu et al., 2020; Yan et al., 2020; Wang et al., 2020) or knowledge of the domains (Zhang et al., 2019).

Instead of using heuristic data augmentations which cannot be used to learn more complex transformations, the desired augmentation can be learnt. The training data can be augmented by transforming samples using a generative model to either come from another part of the domain conditioned on an attribute (Goel et al., 2020), another domain (Zhou et al., 2020), be domain agnostic (Carlucci et al., 2019), or to have a different style (Gowal et al., 2020; Geirhos et al., 2019). These methods often build on work in image generation, such as CycleGAN (Zhu et al., 2017) and StyleGAN (Karras et al., 2019).

We find that these approaches can be effective to learn invariance to the heuristic or learned transformation. However, performance depends on whether the augmentation is a useful property to learn invariance against, else the model may waste capacity. When learning the augmentation, performance is limited by the quality of the learned generative model.

Domain generalization. While many works can be viewed as aiming to improve robustness of models in new domains, here we focus on those methods that aim to learn domain invariant features in stochastically trained machine learning models. One impactful approach to learning features invariant to the domain is DANN (Ganin et al., 2016) (domain adversarial neural networks). This work uses an adversarial network (Goodfellow et al., 2014) to enforce that the features cannot be predictive of the domain. Later work considered a number of ways to enforce invariance, such as the following: minimizing the maximum mean discrepancy (Long et al., 2015; 2017), invariance of the conditional distribution (Li et al., 2018), and invariance of the covariance matrix of the feature distribution (Sun & Saenko, 2016). However, enforcing invariance is challenging and often too strict (Johansson et al., 2019; Zhao et al., 2019). As a result, IRM (Arjovsky et al., 2019) instead enforces that the optimal classifier for different domains is the same. We find that DANN performs consistently best but that performance varies over different datasets and distribution shifts.

Adaptive approaches. Instead of learning a single model and treating samples similarly, adaptive approaches can be used to either modify model parameters or dynamically modify training if there are multiple domains (or groups with different attributes) within the training set. Ways to adapt the model parameters to a new domain include meta learning (as in MAML (Finn et al., 2017)) and adapting the batch normalisation statistics based on the different domains (as in BN-Adapt (Schneider et al., 2020)). Instead of adapting the parameters, other approaches reweigh the importance of samples on which the model struggles. This, intuitively, should force the model to spend more capacity on more challenging parts of the domain (or groups with fewer samples). In GroupDRO (Sagawa et al., 2020), this amounts to putting more mass on samples from the more challenging domains at train time using the loss to determine the challenging domains. In JTT (Liu et al., 2021), this is done by two stage training. First, a classifier is trained in a standard manner. Then, a second classifier is trained by upweighting the samples with a high loss according to the first classifier. The authors posit that the most challenging samples will be those coming from groups with few samples (the low-data regions). We find that neither JTT nor BN-Adapt consistently perform better than the baselines.

Representation learning. Finally, instead of focusing on the downstream task, another approach is to learn a representation $p(z|\boldsymbol{I})$ that approximates the true prior over the latent factors. This can be done by pretraining on large amounts of data such as ImageNet (Russakovsky et al., 2015; Deng et al., 2009), as explored by (Hendrycks et al., 2021) for domain generalization, such that the

learned representation is potentially more robust and general. Our results corroborate their findings: in many cases pretraining is helpful. However, this is not universally true. On CAMELYON17 and IWILDCAM, pretraining did not improve performance across all shifts. For example, pretraining was unhelpful under spurious correlation on CAMELYON17.

Another approach is to learn a representation subject to constraints. This is what the disentanglement literature aims to do: find a representation that is sparse and low dimensional, which hopefully will correspond to a higher level, disentangled representation. Disentanglement is a large research area, so we briefly mention some formative works, such as the β-VAE (Higgins et al., 2017a; Burgess et al., 2017) and FactorVAE (Kim & Mnih, 2018), which build on the original VAE (Kingma & Welling, 2013; Rezende et al., 2014) formulation. Other approaches build on GANs (Goodfellow et al., 2014), such as InfoGAN (Chen et al., 2016). A recent study of these methods (Locatello et al., 2019) found that they did not reliably disentangle the representation into a semantically meaningful set of latent variables. Therefore, it is unclear how robust these methods are when used for distribution shifts, but our results imply that more research is needed to make these approaches practically useful for robustness.

D DATASET

Here we give further details about the datasets in appendix D.1, how we set up the shifts and conditions for these datasets in appendix D.2 and finally we give samples from the different datasets and shifts in appendix D.3.

D.1 DETAILS

Here we give further detail about each of the datasets and how we set the two attributes: y^l, y^a.

DSPRITES. DSPRITES (Matthey et al., 2017) consists of shapes (heart, ellipse, and square), which we augment with three colors (red, green, and blue). The shapes vary in location, scale, and orientation. We make the shape the label, and the attribute the color (we consider other choices in appendix B.3).

MPI3D. MPI3D (Gondal et al., 2019) consists of real images of shapes on a robotic arm. There are six shapes and the images vary in terms of the object color, object size, camera height, background color and the rotation about the horizontal and vertical axis. We make the shape the label and object color the other attribute.

SHAPES3D. SHAPES3D (Burgess & Kim, 2018) consists of images of shapes centered in a synthetic room. There are three shapes and the images vary in terms of the floor hue, object hue, orientation, scale, shape, and wall hue. We make the shape the label and object color the other attribute.

SMALLNORB. SMALLNORB (LeCun et al., 2004) consists of images of toys of five categories with varying lighting, elevation and azimuths. The aim is to create methods that generalize to unseen samples of the five categories. We make the object category the label and azimuth the other attribute for *low-data drift* and *unseen data shift*. We make the lighting the other attribute for *spurious correlation* and under the noise and fixed data size conditions.

CAMELYON17. CAMELYON17 (Koh et al., 2020; Bandi et al., 2018) contains tumour cell images coming from different hospitals. We follow Koh et al. (2020) and make the label the presence of tumor cells and the other attribute the hospital from which the image came from. In the *spurious correlation, low-data drift* settings, we resplit the dataset randomly (there are no held out hospitals). In the *spurious correlation* setting, we correlate the presence of tumor cells with the hospital (all images from a given hospital either do or do not have tumors). In the *low-data* setting, we select the hospitals that are out of distribution for Koh et al. (2020) as the hospitals for which we only have N samples. In the *unseen data shift*, we use the split given by Koh et al. (2020) and optionally use either the in-distribution or out-of-distribution validation set for model selection.

iWildCam. iWildCam (Koh et al., 2020; Beery et al., 2018) contains camera trap imagery. There are a set of locations. The camera is kept in the same fixed spot in each location, and takes photos at different times. The task is to determine if there is an animal in the image and which animal is present. In the *spurious correlation, low-data drift* settings, we resplit the dataset randomly (there are no held out locations). In the *spurious correlation* setting, we correlate the presence of a given animal with a location (all images from a given location *only* show a given animal). In the *low-data drift* setting, we select the locations that are out of distribution for Koh et al. (2020) as the locations for which we only have N samples. In the *unseen data shift* setting, we use the split given by Koh et al. (2020) and optionally use either the in-distribution or out-of-distribution validation set for model selection.

D.2 Evaluated shifts and conditions

We further describe how we set up the shifts in this section and define each of the shifts for each dataset in table 2. Note that the amount of correlation, total number of samples from the low-data region, and the probability of sampling from the low-data or correlated distributions are controllable in our framework. Additionally, we can control the amount of label noise and total dataset size.

Shift 1: Spurious correlation. Under spurious correlation, we correlate y^l, y^a. At test time, these attributes are uncorrelated. We vary the amount of correlation by creating a new dataset with all samples from the correlated distribution in the dataset and N samples from the uncorrelated distribution; this forms the training set. We set $N >= 1$ (as if $N = 0$, then the problem is ill defined as to what is the correct label). The test set is composed of sampling from the uncorrelated distribution and is disjoint from the training samples.

Shift 2: Low-data drift. Under low-data drift, we consider the set $\mathbb{A}^a$. For some subset $\mathbb{A}^a_c \subset \mathbb{A}^a$, we only see N samples of those attributes. For all other values of y^a ($\mathbb{A}^a / \mathbb{A}^a_c$), the model has access to all samples.

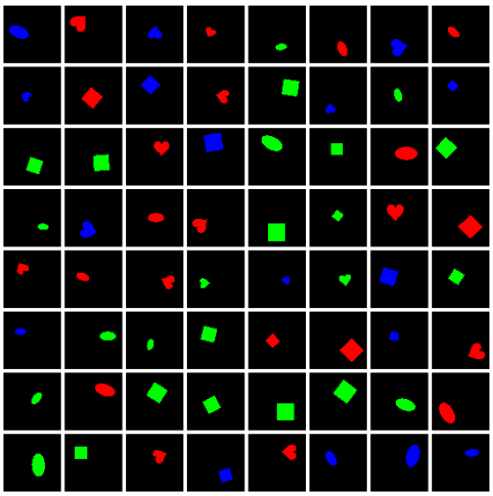
Shift 3: Unseen data shift This is a special case of *low-data drift*, where we set $N = 0$.

Condition 1: Noisy labels. To investigate how methods perform in the presence of noise, we add uniform label noise with increasing probability. We take the low-data setting and fix the value of N. We then vary the amount of noise p.

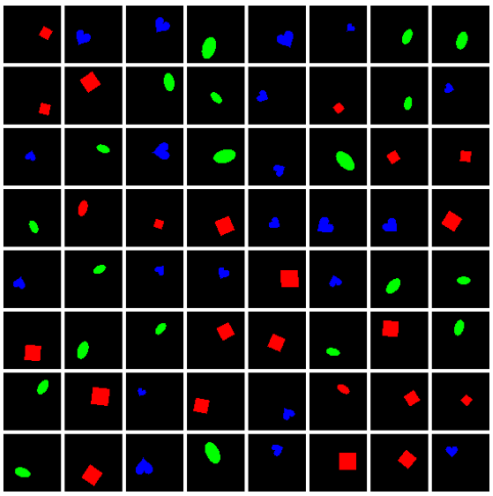
Condition 2: Dataset size. We investigate how performance degrades as the total size of the training dataset changes. We again take the low-data setting but we vary the total number n of samples from the train set and fix the ratio $\frac{N}{n}$.

D.3 Samples from the different distributions

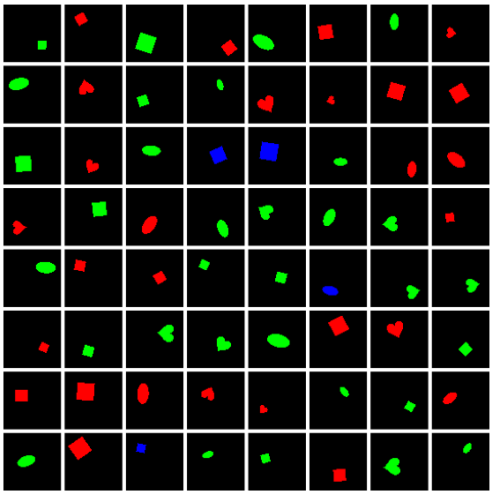
We include samples for each distribution for each dataset in figure 13-18.



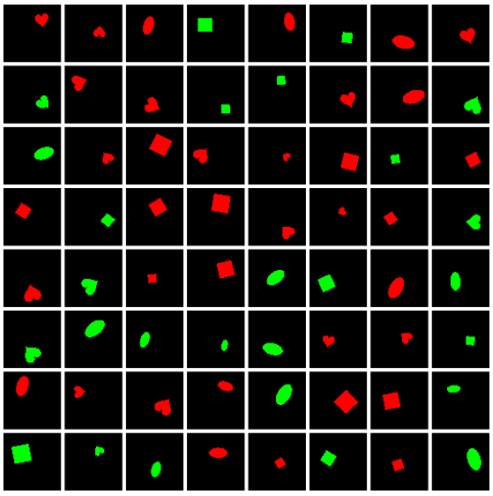
DSPRITES (IID test).



DSPRITES (**Shift 1: Spurious correlation.**)

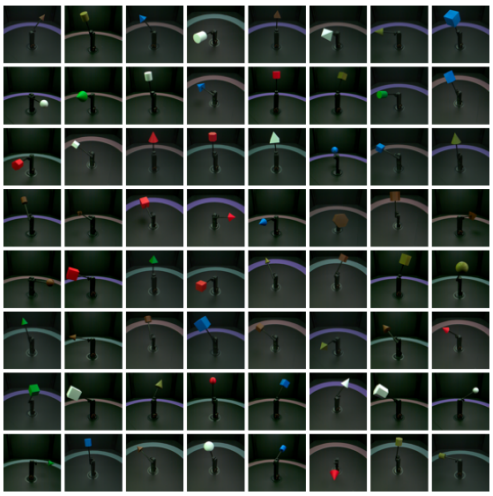


DSPRITES (**Shift 2: Low-data drift.**)

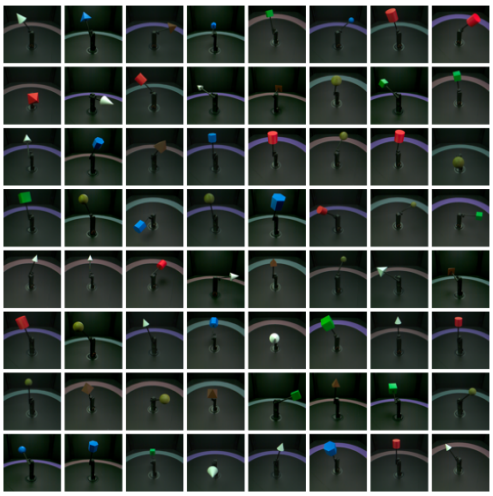


DSPRITES (**Shift 3: Unseen data shift.**)

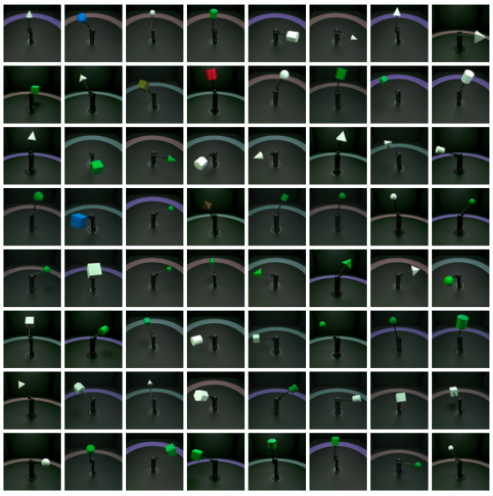
Figure 13: Sample distributions on DSPRITES.



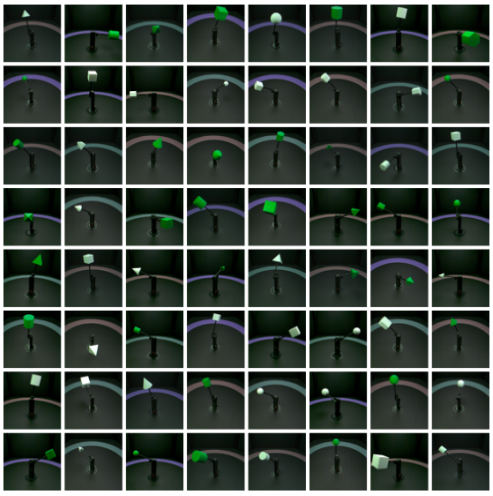
MPI3D (IID test).



MPI3D (**Shift 1: Spurious correlation.**)

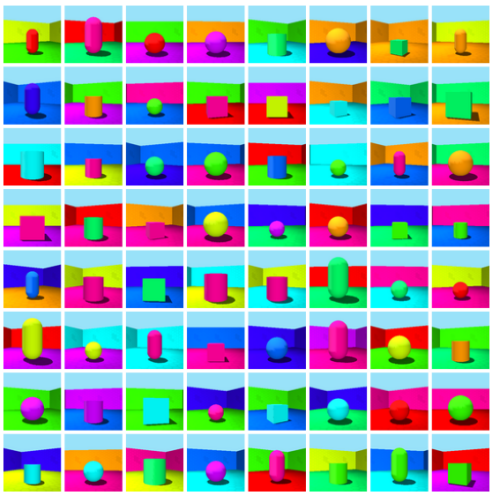


MPI3D (**Shift 2: Low-data drift.**)

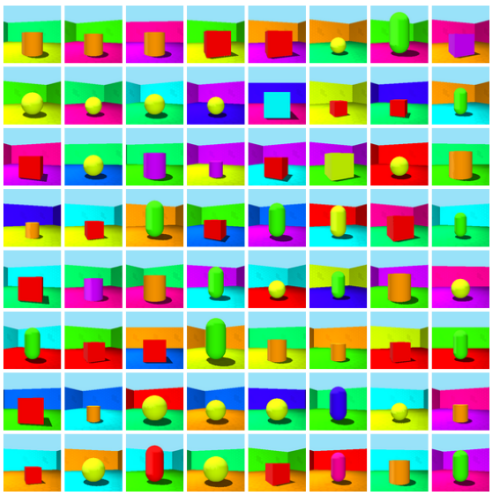


MPI3D (**Shift 3: Unseen data shift.**)

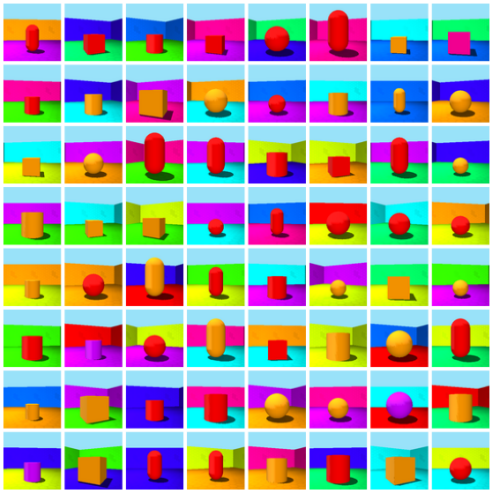
Figure 14: Sample distributions on MPI3D.



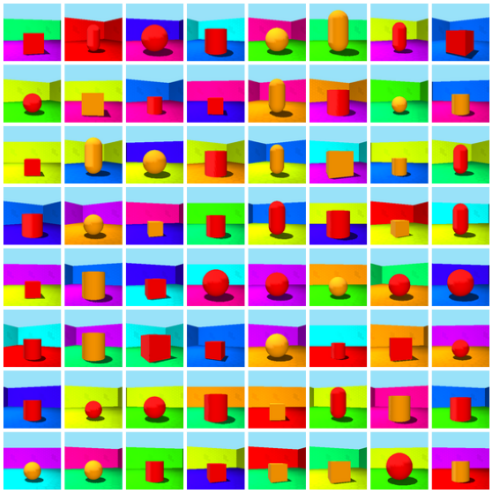
SHAPES3D (IID test).



SHAPES3D (**Shift 1: Spurious correlation.**)

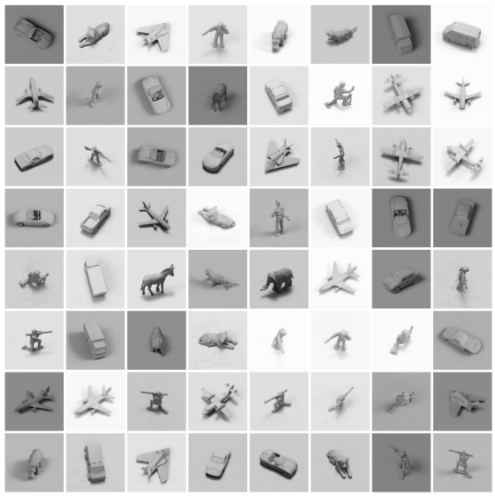


SHAPES3D (**Shift 2: Low-data drift.**)



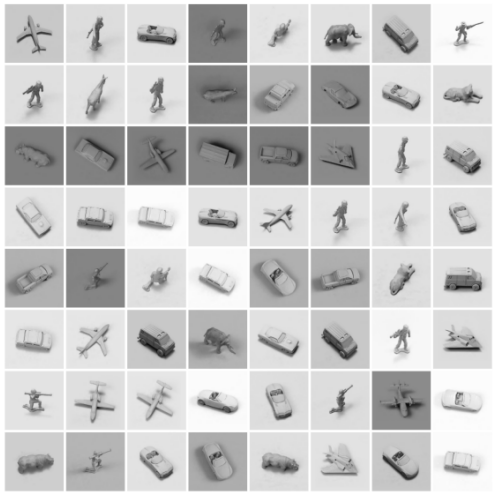
SHAPES3D (**Shift 3: Unseen data shift.**)

Figure 15: Sample distributions on SHAPES3D.

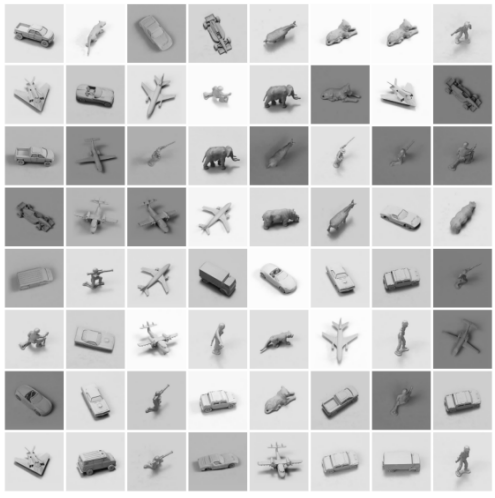


SMALLNORB (IID test).

SMALLNORB (**Shift 1: Spurious correlation.**)

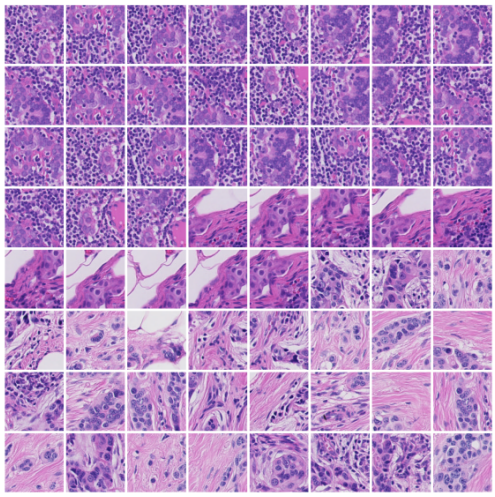


SMALLNORB (**Shift 2: Low-data drift.**)

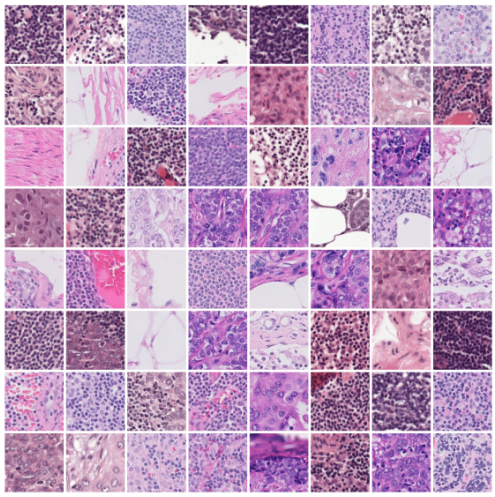


SMALLNORB (**Shift 3: Unseen data shift.**)

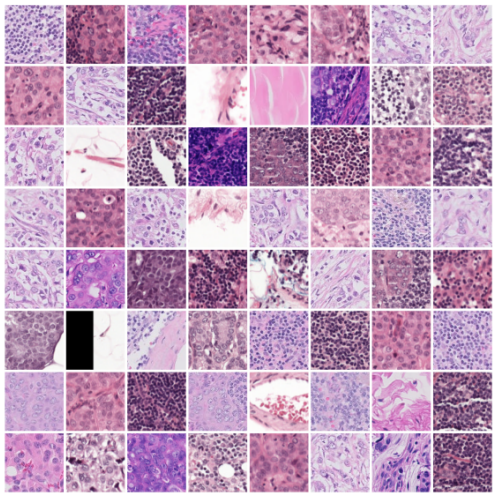
Figure 16: Sample distributions on SMALLNORB.



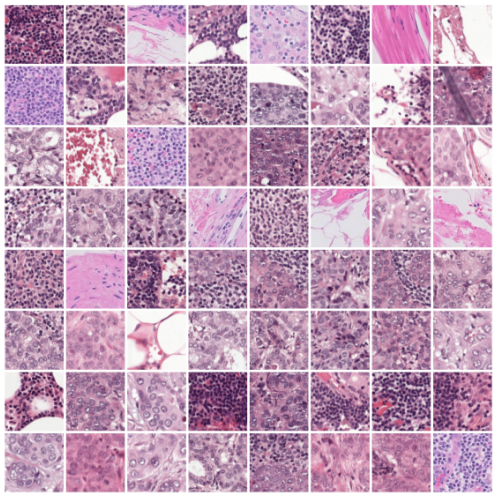
CAMELYON17 (OOD test).



CAMELYON17 (**Shift 1: Spurious correlation.**)

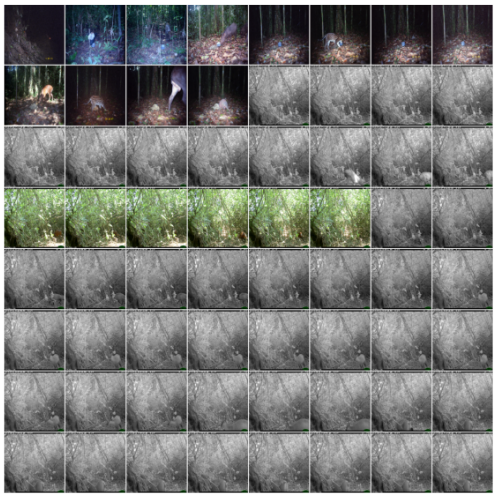


CAMELYON17 (**Shift 2: Low-data drift.**)

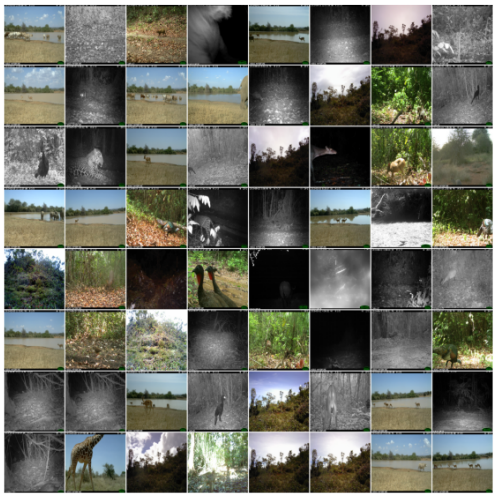


CAMELYON17 (**Shift 3: Unseen data shift.**)

Figure 17: Sample distributions on CAMELYON17.

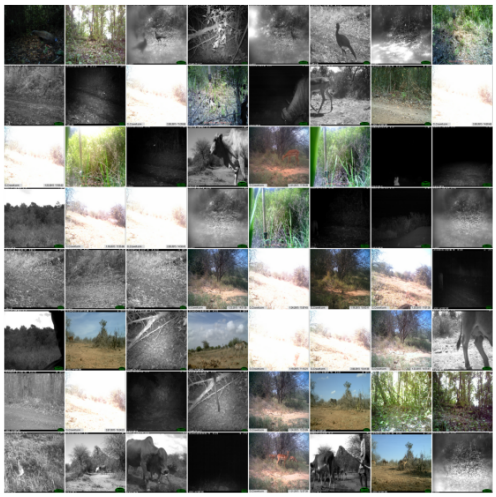


iWildCam (OOD test).



iWildCam (**Shift 1: Spurious correlation.**)

iWildCam (**Shift 2: Low-data drift.**)



iWildCam (**Shift 3: Unseen data shift.**)

Figure 18: Sample distributions on iWildCam.

Dataset	label	nuisance attr	SC	LD / UDS ($\mathbb{A}_c^a$)	Noise (N)	Dataset size (N/n)
DSPRITES	l = shape $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 3$	a = color $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 3$	$v_i \sim u_i$	{blue}	10	0.001
MPI3D	l = shape $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 6$	a = color $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 6$	$v_i \sim u_i$	$\{u_i \mid i > 2\}$	10	0.001
SHAPES3D	l = shape $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 4$	a = obj. color $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 10$	$v_i \sim u_i, i \leq 4$	$\{u_i \mid i > 2\}$	10	0.001
SMALLNORB	l = category $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 5$	a = azimuth $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 18$ b = lighting $w_i \in \mathbb{A}^b$ $\lvert\mathbb{A}^b\rvert = 6$	$v_i \sim w_i, i \leq 5$	$\{u_i \mid i > 4\}$	10	0.001
CAMELYON17	l = tumor $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 2$	a = hospital $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 5$	$v_i = 0 \sim u_{i\in[1,2,3]}$ $v_i = 1 \sim u_{i\in[4,5]}$	$\{u_i \mid i \in \{2, 3\}\}$	10	0.001
IWILDCAM	l = animal $v_i \in \mathbb{A}^l$ $\lvert\mathbb{A}^l\rvert = 186$	a = location $u_i \in \mathbb{A}^a$ $\lvert\mathbb{A}^a\rvert = 324$	$v \sim u$	$\lvert\mathbb{A}_c^a\rvert = 79$	10	0.001

Table 2: The precise dataset shifts we use for each dataset. We describe the label and nuisance attribute for each dataset. We additionally describe the setup for *spurious correlation (SC)* setting and for the *low-data drift (LD)/unseen data shift (UDS)* settings. Finally, we give the hyperparameters set in the *LD* setting when evaluating under the two additional conditions: label noise and fixed dataset size. $\sim$ denotes correlation. We denote each value in the attribute sets using 1-based indexing. For IWILDCAM, not all locations have all animals, so we find the most commonly occurring location for each animal and correlate that animal with that location. Additionally, for IWILDCAM and CAMELYON17, we use the OOD set from Koh et al. (2020) as $\mathbb{A}_c^a$. For IWILDCAM, there are 72 locations.

E EXTENDING THE EVALUATION FRAMEWORK

Our evaluation framework is easily extendable. Different methods, datasets, and distribution shifts can be easily added and evaluated. We include code in appendix E.1-E.3, demonstrating how this is done.

E.1 ADDING A NEW DISTRIBUTION SHIFT

Our framework for creating shifts allows for creating complex distribution shifts. We can define distribution shifts using a comparison operator on one or more labels to create a distribution shift. Shifts can then be composed together to create more complex shifts between train and test. This allows the exploration and evaluation of multiple entangled factors of variation. Finally, there is no necessity for methods to make use of the additional attribute information beyond the label, which allows the evaluation of unseen factors of variation (by not allowing methods to use knowledge of a given factor).

Changing the distribution shift simply amounts to updating four values. These values define (1) how to construct the distribution as a combination of other distributions; (2) the total number of samples from each distribution; (3) the probability of sampling from a given distribution; and (4) the probability that the label is corrupted at training time.

Assume a dataset that has the key *center* which takes values in the range [0, 4] and the key *label* which takes values in the set $\{0, 1\}$. The code in algorithm 1 defines the following dataset: with probability 0.1, we sample an example that is either from center 0 or center 1 from the complete dataset; with probability 0.4 we sample from a total of 100 samples that are from center 3 or 4 with label 0; and with probability 0.5, we sample from a total of 10K samples from centers 0, 1, 2. None of the labels are corrupted.

```
  """Given a string "dist1,dist2,...,distN", the string is first divided into the different distributions,
      separated by commas. Each distribution is defined by a string of "key11:val11:comp11^key12:val12:comp12
      |...^keyNk:valNk:compNk". This string is parsed as the OR of several AND statements. The ORs are at the
      top level (denoted by |), and divide into a set of AND statements. The AND values are denoted by ^ and
      operate at the bottom level.  For each "keyij:valij:compij" pairing, keyij is the key in the dataset,
      valij is the value the key is compared against and compij is the tensorflow comparison function (e.g.
      less, less_equal, equal, greater_equal, or greater)."""
config.filter_fn = "center:0:equal|center:1:equal,center:2:greater^label:0:equal,center:3:less_equal"

# num_samples defines the total number of samples from each distribution. 0 denotes that all samples are taken
config.num_samples = [0, 100, 10_000]

# weights defines the probability of sampling from each distribution.
config.weights = [0.1, 0.4, 0.5]

# Label_noise defines the probability that the label is corrupted (with uniform probability).
config.label_noise = [0.0]
```

Listing 1: Setting up a new distribution shift.

E.2 ADDING A NEW DATASET

Adding a new data loader requires implementing three methods: (1) a preprocessing function for how to preprocess a batch; (2) a function to load the data in a tensorflow datasets format; and (3) a function to batch and shuffle this tensorflow dataset. In algorithm 2, we demonstrate how this is done for the SHAPES3D dataset and in algorithm 3 how the config is updated to include a new dataset.

```
"""The preprocessing function to update the image and label key."""
def shapes3d_preprocess(mode = 'train'):
  del mode

  def _preprocess_fn(example):
    example['image'] = tf.image.convert_image_dtype(example['image'], dtype=tf.float32)
    example['label'] = example['label_shape']
    return example
  return _preprocess_fn


"""The tfds loading function."""
def unbatched_load_shapes3d(subset = 'train', valid_size = 10000, test_size = 10000):
  """Loads the 3D Shapes dataset in tfds format without batching."""
  if subset == 'train':
    ds = tfds.load(name='shapes3d', split=tfds.Split.TRAIN).skip(valid_size + test_size)
  elif subset == 'valid':
```

```
    ds = tfds.load(name='shapes3d', split=tfds.Split.TRAIN).skip(test_size).take(valid_size)
  elif subset == 'train_and_valid':
    ds = tfds.load(name='shapes3d', split=tfds.Split.TRAIN).skip(test_size)
  elif subset == 'test':
    ds = tfds.load(name='shapes3d', split=tfds.Split.TRAIN).take(test_size)
  else:
    raise ValueError('Unknown subset: "{}"'.format(subset))
  return ds


def load_shapes3d(batch_sizes, subset = 'train', is_training = True, num_samples = None, preprocess_fn = None,
      transpose = False, valid_size = 10000, test_size = 10000, drop_remainder = True, local_cache = True):
  """Loads the 3D Shapes dataset.

  The 3D shapes dataset is available at https://github.com/deepmind/3d-shapes.
  It consists of 4 different shapes which vary along 5 different axes:
  - Floor hue: 10 colors
  - Wall hue: 10 colors
  - Object hue: 10 colors
  - Scale: How large the object is.
  - Shape: 4 values -- (cube, sphere, cylinder, and oblong).
  - Orientation: Rotates the object around the vertical axis.

  Args:
    batch_sizes: Specifies how to batch examples. I.e., if batch_sizes = [8, 4]
      then output images will have shapes (8, 4, height, width, 3).
    subset: Specifies which subset (train, valid, train_and_valid, or test) to use.
    is_training: Whether to infinitely repeat and shuffle examples (`True`) or
      not (`False`).
    num_samples: The number of samples to crop each individual dataset variant
      from the start, or `None` to use the full dataset.
    preprocess_fn: Function mapped onto each example for pre-processing.
    transpose: Whether to permute image dimensions NHWC -> HWCN (`True`) or not (`False`).
    valid_size: Size of the validation set to take from the training set.
    test_size: Size of the validation set to take from the training set.
    drop_remainder: Whether to drop the last batch(es) if they would not match
      the shapes specified by `batch_sizes`.
    local_cache: Whether to locally cache the dataset.

  Returns:
    ds: Fully configured dataset ready for training/evaluation.
  """
  if preprocess_fn is None:
    preprocess_fn = shapes3d_preprocess
  ds = unbatched_load_shapes3d(subset=subset, valid_size=valid_size, test_size=test_size)
  total_batch_size = np.prod(batch_sizes)
  if subset == 'valid' and valid_size < total_batch_size:
    ds = ds.repeat().take(total_batch_size)
  ds = batch_and_shuffle(ds, batch_sizes, is_training=is_training, transpose=transpose, num_samples=
     num_samples, preprocess_fn=preprocess_fn,  drop_remainder=drop_remainder, local_cache=local_cache)
  return ds
```

Listing 2: Setting up a new dataset.

```
"""Update training data configuration."""
config.data.train_kwargs.load_kwargs = dict()
config.data.train_kwargs.load_kwargs['dataset_loader'] = unbatched_load_shapes3d
config.data.train_kwargs.load_kwargs['dataset_kwargs'] = dict(subset='train')  # Update with data specific
    values as appropriate.
config.data.train_kwargs.load_kwargs['preprocess_fn'] = shapes3d_preprocess

"""Update testing data configuration."""
config.data.test_kwargs = dict()
config.data.test_kwargs['loader'] = load_shapes3d
config.data.test_kwargs['load_kwargs'] = dict(subset='test')
```

Listing 3: Updating the config to use the new dataset.

E.3 Adding a new method

Adding a new loss function. Adding a new loss function simply amounts to extending the abstract class in algorithm 4 and updating the config (see algorithm 5) to point to this loss function.

```
class LearningAlgorithm(hk.Module):
  """Class to encapsulate a learning algorithm."""

  def __init__(self, loss_fn, name, **kwargs):
    """Initializes the algorithm with the given loss function."""
    super().__init__(name=name)
    self.loss_fn = loss_fn

  def __call__(self, logits, targets, reduction, attributes = None):
    """The loss function of the learning algorithm.

    Args:
      logits: The predicted logits input to the training algorithm.
      targets: The ground truth value to estimate.
```

```
      reduction: How to combine the loss for different samples (mean or sum).
      attributes: An optional set of properties of the input data.
    Returns:
      scalars: A dictionary of key and scalar estimates. The key 'loss' is the loss that should be minimized.
      preds: The raw softmax predictions.
    """
    pass

  def adversary(self, logits, attributes, reduction = 'mean', targets = None):
    """The adversarial loss function.

    If la = LearningAlgorithm(), this function is applied in a min-max game with la(). The model is trained to
       minimize the loss arising from la(), while maximizing the loss from the adversary (la.adversary()). The
       adversarial part of the model tries to minimize this loss.

    Args:
      logits: The predicted value input to the training algorithm.
      attributes: A set of attributes of the input data.
      reduction: How to combine the loss for different samples ('mean' or 'sum').
      targets: The ground truth value to estimate (optional).
    Returns:
      scalars: A dictionary of key and scalar estimates. The key 'adv_loss' is the value that should be
      minimized (for the adversary) and maximized (for the model). If empty, this learning algorithm has no
      adversary.
    """
    # Do nothing.
    return {}


class ERM(base.LearningAlgorithm):
  """An example, demonstrating how to compute the empirical risk within our framework."""

  def __init__(self, loss_fn, name = 'empirical_risk'):
    super().__init__(loss_fn=loss_fn, name=name)

  def __call__(self, logits, targets, reduction = 'mean', **unused_kwargs):
    loss = self.loss_fn(logits, targets, reduction=reduction)
    return {'loss': loss}, logits
```

Listing 4: Setting up a new loss function.

```
"""Config for which learning algorithm to use."""
config.learner = dict()
config.learner['fn'] = ERM
config.learner['kwargs'] = dict()  # Update with algorithm specific parameters.
```

Listing 5: Updating the config for the new loss function.

Extending other parts of the framework. We similarly define abstract classes, initialized in the config, for model selection, preprocessing and postprocessing a batch, and adapting model parameters at train and test time. These can all be extended in order to include new models, algorithms, preprocessing and postprocessing steps or adaptive approaches.

F METHOD

Here we give further description of the methods we implement and how they relate to our robustness framework. This allows us to obtain an intuition of what guarantees each method gives in this context and thereby under what circumstances they should promote generalization.

Backbone architecture. We investigate the performance of different standard vision models on the robustness task. The model is trained on p_{train} to predict the true label, making no use of the additional attribute information. We use weighted resampling p_{reweight} (Vapnik, 1992) to oversample from the parts of the distribution that have a lower probability. This is what we refer to as the *standard* setup.

Heuristic data augmentation. In this case, we use standard heuristic augmentation methods in order to augment the training samples in p_{train}. Instead of attempting to learn the conditional generative model, in this approach, we 'fake' the generative model by augmenting the images using a set of heuristic functions to create $p_{\text{aug}}, \alpha = 1$. However, we have to heuristically choose these functions, so the generative model they approximate may *not* correspond to the true underlying generative model $p(\boldsymbol{x}|y^{1:K})$. In practice, these methods make no use of the additional attribute information and are trained to predict the label, as in the standard setup.

Learned data augmentation. Again, we approximate the underlying generative model $p(\boldsymbol{x}|y^{1:K})$ by a set of augmentations. However, instead of heuristically choosing these augmentation functions, we learn them from data. We learn a function that, given an image $\boldsymbol{x}$ and attribute $y^a = v_i$, transforms $\boldsymbol{x}$ to have another attribute value $y^a = v_j$, while keeping all other attributes fixed. We can then use this function to generate new samples $p_{\text{aug}}, \alpha = 1$ with any distribution over y^a. In particular, we generate samples under the uniform distribution. In this case, the additional attribute information is used to generate new samples from a given image. However, the performance of this approach is highly dependent on the quality of the learned generative model. We follow Goel et al. (2020), who use CYCLEGAN (Zhu et al., 2017) to learn how to transform an image with one attribute value to have that of another. (We do not use their additional SGDRO objective as we want to study the impact of the data augmentation process alone.) In our case, there are more than two attribute values, so we use STARGAN (Choi et al., 2018) to learn a single model that, conditioned on an input image and desired attribute, transforms that image to have the new attribute.

Domain generalization. These works were devised to improve domain generalization. They can also be seen as a form of representation learning. The aim is to recover $p(z|\boldsymbol{x})$ such that the representation z is independent of the domain (in our framework, the attribute y^a): $p(y^a, z) = p(y^a)p(z)$. If this is achieved, then the task specific classifier $p(y^l|z)$ will be independent of y^a by definition. However, these approaches rely on the ability of the underlying method to learn invariance.

Adaptive approaches. These works modify the reweighting distribution in p_{reweight} using multi stage training. The models are trained first as for the standard setup, giving a classifier f_o. JTT (Liu et al., 2021) then uses f_o to approximate the difficulty of the sample. The more difficult samples are weighted higher in the second stage according to a factor λ. This is equivalent to sampling the more difficult sample λ times in a batch (thereby learning a more complex function W). BN-Adapt (Schneider et al., 2020) learns $|\mathbb{A}^a|$ models in the second stage by modifying the batch normalization parameters. For the i-th model, $W(y^a = v_i) = 1, W(y^a = v_j) = 0$ for $i \neq j$. However, neither of these methods give strong guarantees on the properties of the final model.

Representation learning. Finally, in representation learning, the aim is to learn an initial representation that has preferable properties to standard ERM training; the motivation for this approach is discussed in appendix 2. To learn the prior, we can pretrain f on large amounts of auxiliary data, such as on ImageNet (Russakovsky et al., 2015). This has been demonstrated to improve model robustness and uncertainty between datasets (Hendrycks et al., 2019), but here we investigate its utility under different distribution shifts. Another approach is to attempt to learn a disentangled representation with a VAE, as in β-VAE (Higgins et al., 2017a), where z would then describe the underlying factors of variation for the generative model. However, the robustness of these methods is dependent on the quality of the learned representation for the specific robustness task.

Model	# Parameters (M)
ResNet18	11.18
ResNet50	23.51
ResNet101	42.51
MLP	3.34
ViT	85.66

Table 3: The number of parameters for each model. While ViT has the largest capacity, it was the most brittle to train and only achieved best performance on iWildCam. On the other datasets, the ResNets performed best.

G Implementation

We first describe the architectures and precise implementation of each approach in appendix G.1-G.6 and give training details in appendix G.7. We then give the sweeps over the hyperparameters considered in appendix G.8. We do not claim that these are the best possible results obtainable with each method (which would require much larger sweeps, quickly becoming computationally infeasable to compare all methods), but they are representative of performance of each approach.

G.1 Base architectures

We train three types of models. These all have different capacities, which we report in table 3.

ResNets. We use the standard ResNet18, ResNet50, and ResNet101 setups (He et al., 2016).

MLP. For the MLP, we use a 4 layer MLP with 256 hidden units.

ViT. For the ViT (Dosovitskiy et al., 2021), we set the parameters as follows. For the smaller 64x64 images (dSprites, MPI3D, Shapes3D), we use a patch size of 4 with a hidden size of 256. For the transformer, we use 512 for the width of the MLP, 8 heads and 8 layers. We use a dropout rate of 0.1. For the medium 96x96 images (Camelyon17, SmallNorb), we use a patch size of 12 and for the 256x256 images (iWildCam), a patch size of 16.

G.2 Heuristic Augmentation

ImageNet Augmentation. ImageNet augmentation is composed of random crops and color jitter. We use the standard ratios as used in ImageNet training (He et al., 2016). We only apply the augmentation to the first three channels of a dataset (replicating across three channels if the data is grayscale).

AugMix (Hendrycks et al., 2020). AugMix composes multiple k sequences of augmentations, randomly sampled from thirteen base augmentations. We use their default: $k = 3$.

RandAugment (Cubuk et al., 2020). RandAugment randomly samples N augmentations from a set sixteen base augmentations with a severity M. For the augmentation parameters (cutout and translate) based on image size, we interpolate between the values used for ImageNet on 224 images and Cifar10 on 32 images. We set $N = 3, M = 5$.

AutoAugment (Cubuk et al., 2019). If the image size is less than 128 (e.g. all datasets but iWildCam), we use the cifar10 policy, else we use the ImageNet policy. Again, for the augmentation parameters (cutout and translate) based on image size, we interpolate between the values used for ImageNet on 224 images and Cifar10 on 32 images.

G.3 Learned Augmentation

CycleGAN (Goel et al., 2020; Choi et al., 2018; Zhu et al., 2017). This approach proceeds in two stages. The first stage learns how to transform images of one attribute to have another attribute. In

this stage, these models are trained to minimize three losses: a reconstruction loss $\mathcal{L}_r$, classifier loss $\mathcal{L}_c$, and adversarial loss $\mathcal{L}_a$. The final loss is a combination of these: $\mathcal{L} = \lambda_r \mathcal{L}_r + \lambda_c \mathcal{L}_c - \lambda_a \mathcal{L}_a$. We set $\lambda_c = 1, \lambda_r = 1$ and we sweep over λ_a. We train this model with a learning rate $\text{lr}_{\text{STARGAN}}$ (using the same learning rate for the generator and discriminator). We use the same ResNet style architecture as CYCLEGAN, except we append to the image a one hot encoding designating the original attribute, as described in STARGAN (Choi et al., 2018). We use five ResNet blocks for images of size 64 (DSPRITES, MPI3D, SHAPES3D), six ResNet blocks for images of size 96 (CAMELYON17, SMALLNORB) and nine ResNet blocks for images of size 256 (IWILDCAM).

The second stage uses the pretrained STARGAN model in order to obtain p_{aug}. For each image, we uniformly at random select a value for y^a. We then transform the image to have the new attribute value (while keeping the label fixed). We then use this image as an augmented sample. We set $\alpha = 0$, using no real samples.

G.4 DOMAIN GENERALIZATION

IRM (Arjovsky et al., 2019). IRM enforces that the optimal classifier for all domains is the same. This is done using two terms in the risk: a term to minimize the overall risk and a term to enforce the constraint, with a tradeoff λ.

DeepCORAL (Sun & Saenko, 2016). DeepCORAL enforces that the mean and covariance of the learned features in different domains is the same. This is achieved by two terms in the loss: a term to minimize the overall risk and a term to penalize differences in the mean and covariance across each pair of domains. These are weighted according to a tradeoff λ.

Domain MixUp (Gulrajani & Lopez-Paz, 2021). Domain MixUp enforces smoothness in the label prediction by interpolating between images from different domains and enforcing the prediction similarly interpolates between the labels. We set the interpolation parameter $\lambda = 0.2$ (as in MixUp Zhang et al. (2018)).

DANN (Ganin et al., 2016). DANN trains an adversary g that attempts to predict the attribute from the learned representation and the model seeks to fool this adversary while also minimizing downstream accuracy: $\mathcal{L} = \mathcal{L}_c(l(z)) - \lambda_a \mathcal{L}_a(g(z))$. For the adversary, we use a two layer MLP with ReLUs (Nair & Hinton, 2010) and hidden sizes of size 64.

SagNet (Nam et al., 2021). SagNet aims to learn a representation z that is invariant to style by using an adversary. The classification loss $\mathcal{L}_c$ aims to use the content predictor c to classify z. The adversary $\mathcal{L}_a$ aims to use the style predictor s to predict the class. This gives a final loss: $\mathcal{L} = \mathcal{L}_c(c(z)) - \lambda_a \mathcal{L}_a(s(z))$. We use the feature extractor of a ResNet18 (He et al., 2016) (before the average pool) as the representation z. For the content c and style predictor s, we use MLPs with three hidden dimensions of size 64 and ReLUs. We set $\lambda_a = 1$.

G.5 ADAPTIVE APPROACHES

JTT (Liu et al., 2021). JTT uses a pretrained classifier to find the most challenging samples. It then reweights these samples by λ. We set $\lambda = 20$ and train the first model for half the training time.

BN-Adapt (Schneider et al., 2020). BN-Adapt adapts the parameters of the original model according to the attribute label using a hyperparameter r (N/n in their paper). Intuitively, this controls the the strength of the original model r and the new model 1. We set $r = 10$.

G.6 REPRESENTATION LEARNING

β-VAE (Higgins et al., 2017a). β-VAE poses the original VAE objective using constrained optimisation to obtain a new objective which has a tradeoff β between the log-likelihood objective and the KL objective. We resize images to 64x64 and use a convolutional architecture as in Higgins et al. (2017a). The encoder consists of the following layers (with ReLUs) to obtain the final representation z of size L: Conv 16x5x5 (stride 1, spatial feature size: 60), Conv 32x4x4 (stride 2, spatial feature

size 29), Conv 64x3x3 (stride 1, spatial feature size 27), Conv 128x3x3 (stride 2, spatial feature size 13), Conv 256x4x4 (stride 1, spatial feature size 10), Conv 512x4x4 (stride 2, spatial feature size 4), Conv 512x4x4 (stride 1, spatial feature size 1), Linear L. The decoder has the reverse setup to the encoder with convolutional transpose layers. The learned representation is kept fixed when training the downstream classifier.

There are two terms to trade-off: the strength of β and the size of the latent representation L. However, a larger latent will need a stronger β. We use the $\beta_{\text{norm}} = \frac{\beta L}{I}$ which normalises these values by the data size I. We then sweep over L and β_{norm}.

Finally, to train the downstream classifier, we fix the encoder of the β-VAE. We then train an MLP (the same as used in appendix G.1) from the representation of size L to the labels.

Pretrained on ImageNet. We resize images to 224x224 and pass them to a model pretrained on ImageNet. We finetune the full network.

G.7 TRAINING.

We train the ResNets and MLP with the Adam optimizer for a maximum of 100K steps on the synthetic datasets and 200K steps on CAMELYON17 and iWILDCAM with a batch size of 128. The ViT is trained with a batch size of 1024 (it did not converge for smaller batch sizes). All models are trained with early stopping using the validation accuracy.

G.8 SWEEPS.

The sweeps are given in table 4.

Method	Sweeps
ResNets	learning rate: [1e-2, 1e-3, 1e-4]
MLP	learning rate: [1e-2, 1e-3, 1e-4]
ViT	learning rate: [1e-2, 1e-3, 1e-4]
ImageNet Augm	learning rate: [1e-2, 1e-3, 1e-4]
AugMix	learning rate: [1e-2, 1e-3, 1e-4]
RandAugment	learning rate: [1e-2, 1e-3, 1e-4]
AutoAugment	learning rate: [1e-2, 1e-3, 1e-4]
CycleGAN	learning rate: [1e-4] λ_a: [0.01, 0.1, 1, 10] $\text{lr}_{\text{StarGAN}}$: [1e-3, 1e-4, 1e-5]
IRM	learning rate: [1e-2, 1e-3, 1e-4] λ: [0.01, 0.1, 1, 10]
MixUp	learning rate: [1e-2, 1e-3, 1e-4]
CORAL	learning rate: [1e-2, 1e-3, 1e-4] λ: [0.01, 0.1, 1, 10]
SagNet	learning rate: [1e-2, 1e-3, 1e-4]
DANN	learning rate: [1e-2, 1e-3, 1e-4] λ_a: [0.01, 0.1, 1, 10]
JTT	learning rate: [1e-2, 1e-3, 1e-4]
BN-Adapt	learning rate: [1e-2, 1e-3, 1e-4]
β-VAE	learning rate: [1e-4] $\log(\beta_{\text{norm}})$: linspace(1e-4,10,4) L : linspace(10,210,4)
Pretrained on ImageNet	learning rate: [1e-2, 1e-3, 1e-4]

Table 4: The sweeps over each method for each of the five seeds. The maximum total sweep size (due to capacity) is eight models per seed. For each seed, the best model (based on the ID or OOD validation set) is selected from the hyperparameter sweep and used at evaluation to compute the test time performance.