A Detailed Descriptions of Meta-Learning Approaches

For ARM-CML, we introduce two neural networks: a context network $f_{\text{cont}}(\cdot\,;\phi) : \mathcal{X} \to \mathbb{R}^D$, as mentioned in subsection 3.3, and a *prediction network* $f_{\text{pred}}(\cdot\,,\cdot\,;\theta) : \mathcal{X} \times \mathbb{R}^D \to \mathcal{Y}$, parameterized by θ. As discussed, f_{cont} processes each example $\mathbf{x}_k$ in the mini batch separately to produce contexts $\mathbf{c}_k \in \mathbb{R}^D$ for $k = 1, \ldots, K$, which are averaged together into $\mathbf{c} = \frac{1}{K}\sum_{k=1}^{K} \mathbf{c}_k$. In our experiments, we choose D to be the dimensionality of $\mathbf{x}$, such that we can concatenate each $\mathbf{x}_k$ and $\mathbf{c}$ along the channel dimension to produce the input to f_{pred}. Thus, f_{pred} processes each $\mathbf{x}_k$ separately to produce an estimate of the output $\hat{y}_k$, but it additionally receives $\mathbf{c}$ as input. In this way, f_{cont} can provide information about the entire batch of K unlabeled data points to f_{pred} for predicting the correct outputs.

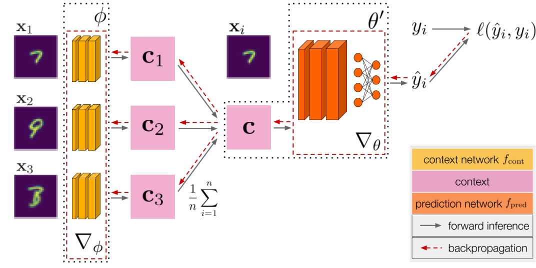

Figure 4: During inference for ARM-CML, the context network produces a vector $\mathbf{c}_k$ for each input image $\mathbf{x}_k$ in the batch, and the average of these vectors is used as the context $\mathbf{c}$ is input to the prediction network. This context may adapt the model by providing helpful information about the underlying test distribution, and this adaptation can aid prediction for difficult or ambiguous examples. During training, we compute the loss of the post adaptation predictions and backpropagate through the inference procedure to update the model.

A schematic of this method is presented in Figure 4. The post adaptation model parameters θ' are $[\theta, \bar{c}]$. Since we only ever use the model after adaptation, both during training and at test time, we can simply define $g(\mathbf{x}; \theta') = f_{\text{pred}}(\mathbf{x}, \mathbf{c}; \theta)$, leaving the model's behavior before adaptation undefined. We then also see that h is a function that takes in $(\theta, \mathbf{x}_1, \ldots, \mathbf{x}_K)$ and produces $\left[\theta, \frac{1}{K}\sum_{k=1}^{K} f_{\text{cont}}(\mathbf{x}_k; \phi)\right]$. In the streaming setting tested in subsection 4.4, we keep track of the average context over the previous test points $\mathbf{c}$ and we maintain a counter t of the number of test points seen so far.[1] When we observe a new point $\mathbf{x}$, we increment the counter and update the average context as $\frac{t}{t+1}\mathbf{c} + \frac{1}{t+1} f_{\text{cont}}(\mathbf{x}; \phi)$, and then we make a prediction on $\mathbf{x}$ using this updated context. Notice that, with this procedure, we do not need to store any test points after they are observed, and this procedure results in an equivalent context to ARM-CML in the batch test setting after observing K data points.

At a high level, ARM-BN operates in a similar fashion to ARM-CML, thus we group these methods together into the umbrella of contextual approaches. However, most of the details are different. For ARM-BN, there is no context network, and h has no parameters, i.e., ϕ is empty. The model g is again specified via a prediction network f_{pred}, which must have batch normalization layers. Batch normalization typically tracks a running average of the first and second moments of the activations in these layers, which are then used at test time. Thus, we can view these moments, along with the weights in f_{pred}, as part of θ. ARM-BN instead defines h to swap out these moments for the moments computed via the activations on the test batch. This method is remarkably simple, and in deep learning libraries such as PyTorch [48], the implementation requires the changing of a single line of code. However, as shown in Section 4, this method also performs very well empirically, and it is further boosted by meta-training.

In the streaming setting, ARM-BN is also similar to ARM-CML, however it is slightly more complex due to the requirement of computing second moments. Denote the context after seeing t test points as $\mathbf{c} = [\boldsymbol{\mu}, \boldsymbol{\sigma}^2]$, the mean and variance of the batch normalization layer activations on the points so far. Upon seeing a new test point, let $\mathbf{a}$ denote the batch normalization layer activations computed from this new point, with size h. We then update the new context to be $\left[\frac{ht}{h(t+1)}\boldsymbol{\mu} + \frac{\sum \mathbf{a}}{h(t+1)}, \frac{ht}{h(t+1)}(\boldsymbol{\sigma}^2 + \boldsymbol{\mu}^2) + \frac{\sum \mathbf{a}^2}{h(t+1)} - (\frac{ht}{h(t+1)}\boldsymbol{\mu} + \frac{\sum \mathbf{a}}{h(t+1)})^2\right]$. Again note that we do not store any test points and that we arrive at the same context as the batch test setting after observing K data points.

[1] An alternative to maintaining a counter t is to use an exponential moving average, though we do not experiment with this option.

Finally, for ARM-LL, we note that ϕ contains only the parameters of the loss network f_{loss}, and h is defined as

$$h(\theta, \mathbf{x}_1, \ldots, \mathbf{x}_K; \phi) = \theta - \alpha \nabla_\theta \| [f_{\text{loss}}(g(\mathbf{x}_1; \theta); \phi), \ldots, f_{\text{loss}}(g(\mathbf{x}_K; \theta); \phi)] \|_2 .$$

We found that $\alpha = 0.1$ worked well for our experiments. We used 1 gradient step for both meta-training and meta-testing. Finally, though we did not evaluate ARM-LL in the streaming setting, in principle this method can be extended to this setting by performing a single gradient step with a smaller α after observing each test point. In an online fashion, we can continually update the model parameters over the course of testing rather than initializing from the meta-learned parameters for each test point.

B Additional Experimental Details

When reporting our results, we run each method across three seeds and report the mean and standard error across seeds. Standard error is calculated as the sample standard deviation divided by the square root of 3. We checkpoint models after every epoch of training, and at test time, we evaluate the checkpoint with the best worst case validation accuracy. Training hyperparameters and details for how we evaluate validation and test accuracy are provided for each experimental domain below. All hyperparameter settings were selected in preliminary experiments using validation accuracy only.

We also provide details for how we constructed the training, validation, and test splits for each dataset. These splits were designed without any consideration for the train, validation, and test accuracies of any method. All of these design choices were made either intuitively – such as maintaining the original data splits for MNIST – or randomly – such as which users were selected for which splits in FEMNIST – or with a benign alternate purpose – such as choosing disjoint sets of corruptions.

B.1 Rotated MNIST details

We construct a training set of 32292 data points by replicating 90% of the original training set – separating out a validation set – and then applying random rotations to each image. The rotations are not dependent on the image or label, but certain rotations are sampled much less frequently than others. In particular, rotations of 0 through 20 degrees, inclusive, have 7560 data points each, 30 through 50 degrees have 2160 points each, 60 through 80 have 648, 90 through 110 have 324 each, and 120 to 130 have 108 points each.

We train all models for 200 epochs with mini batch sizes of 50. We use Adam updates with learning rate 0.0001 and weight decay 0.0001. We construct an additional level of mini batching for our method as described in subsection 3.3, such that the batch dimensions of the data mini batches is 6×50 rather than just 50, and each of the inner mini batches contain examples from the same rotation. We refer to the outer batch dimension as the *meta batch size* and the inner dimension as the batch size. All methods are still trained for the same number of epochs and see the same amount of data. Finally, DRNN uses an additional learning rate hyperparameter for their robust loss, which we set to 0.01 across all experiments [55].

Due to the large number of groups in this setting, we only compute validation accuracy every 10 epochs. When computing validation accuracy, we estimate accuracy on each rotation by randomly sampling 300 of the held out 6000 original training points and applying the specific rotation, resampling for each validation evaluation. This is effectively the same procedure as the test evaluation, which randomly samples 3000 of the 10000 test points and applies a specific rotation.

We retain the original $28 \times 28 \times 1$ dimensionality for the MNIST images, and we divide inputs by 256. We use convolutional neural networks for all methods with varying depths to account for parameter fairness. For ERM, the UW baseline, and DRNN, the network has four convolution layers with 128 filters of size 5×5, followed by 4×4 average pooling, one fully connected layer of size 200, and a linear output layer. Rectified linear unit (ReLU) nonlinearities are used throughout, and batch normalization [26] is used for the convolution layers. The first two convolution layers use padding to preserve the input height and width, and the last two convolution layers use 2×2 max pooling. For our method and context ablation, we remove the first two convolution layers for the prediction network, but we incorporate a context network. The context network uses two convolution layers with 64 filters of size 5×5, with ReLU nonlinearities, batch normalization, and padding, followed

by a final convolution layer with padding. This last layer has number of filters, of size 5×5, equal to 12 in the case of MNIST, 3 for CIFAR and Tiny ImageNet-C, and 1 for FEMNIST.

B.2 FEMNIST details

FEMNIST, and EMNIST in general, is a significantly more challenging dataset compared to MNIST due to its larger label space (62 compared to 10 classes), label imbalance (almost half of the data points are digits), and inherent ambiguities (e.g., lowercase versus uppercase "o") [9]. In processing the FEMNIST dataset,[2] we filter out users with fewer than 100 examples, leaving 262, 50, and 35 unique users and a total of 62732, 8484, and 8439 data points in the training, validation, and test splits, respectively. The smallest users contain 104, 119, and 140 data points, respectively. We keep all hyperparameters the same as MNIST, except we set the meta batch size for our method to be 2.

We additionally compare to q-FedAvg on this domain, as this method is specifically designed for federated learning settings [34]. We modify the authors' publicly available code[3] to run experiments in our setting, and we will make this fork available upon publication along with our own code base. This method follows its own update rule and hyperparameter settings, and we separately optimize the hyperparameters for q-FedAvg as described in Li et al. [34]. Specifically, we first set $q = 0$ and sweep learning rate values between 0.0001 and 1.0, and then we sweep $q \in \{0.001, 0.01, 0.1, 0.5, 1, 2, 5, 10, 15\}$ with the optimal learning rate. With this procedure, we set learning rate to be 0.8 and q to be 0.001.

We compute validation accuracy every epoch by iterating through the data of each validation user once, and this procedure is the same as test evaluation. Note that all methods will sometimes receive small batch sizes as each user's data size may not be a multiple of 50, and though this may affect ARM methods, we demonstrate in subsection 4.4 that ARM-CML and ARM-BN can adapt using batch sizes much smaller than 50. The network architectures are the same as the architectures used for rotated MNIST.

B.3 CIFAR-10-C and Tiny ImageNet-C details

For both CIFAR-10-C and Tiny ImageNet-C, we construct training, validation, and test sets with 56, 17, and 22 groups, respectively. Each group is based on type and severity of corruption. We split groups such that corruptions in the training, validation, and test sets are disjoint. Specifically, the training set consists of Gaussian noise, shot noise, defocus blur, glass blur, zoom blur, snow, frost, brightness, contrast, and pixelate corruptions of all severity levels. Similarly, the validation set consists of speckle noise, Gaussian blur, and saturate corruptions, and the test set consists of impulse noise, motion blur, fog, and elastic transform corruptions of all severity levels. For two corruptions, spatter and JPEG compression, we include lower severities (1-3) in the training set and higher severities (4-5) in the validation and test sets. For the training and validation sets, each group consists of 1000 images for CIFAR-10-C and 2000 images for Tiny ImageNet-C, giving training sets of size 56000 and 112000, respectively. We use the full test set of 10000 images for each group, giving a total of 220000 test images for both CIFAR-10-C and Tiny ImageNet-C.

In these experiments, we train ResNet-50 [19] models with a support size of 50 and meta batch size of 6. As described above, the context ablation and ARM-CML additionally use small convolutional context networks, and the learned loss ablation and ARM-LL use small fully connected loss networks. The images are normalized by the ImageNet mean and standard deviation before they are passed through the model. For CIFAR-10-C, we train models from scratch for 100 epochs, and for Tiny ImageNet-C we fine tune a pretrained model for 50 epochs. We use stochastic gradient descent with learning rate 0.01, momentum 0.9, and weight decay 0.0001. We evaluate validation accuracy after every epoch and perform model selection based on the worst case accuracy over groups. We perform test evaluation by randomly sampling 3000 images from each group and computing worst case and average classification accuracy across groups.

[2] https://github.com/TalwalkarLab/leaf/tree/master/data/femnist.
[3] https://github.com/litian96/fair_flearn

C Additional Experiments

Unknown groups. In the case of unknown groups, one option is to use unsupervised learning techniques to discover group structure in the training data. To test this option, we focus on rotated MNIST and ARM-CML, which performs the best on this dataset, and train a variational autoencoder (VAE) [30, 53] with discrete latent variables [27, 42] using the training images and labels. We define the latent variable, which we denote as c to differentiate from the group z, to be Categorical with 12 possible discrete values, which we purposefully choose to be smaller than the number of rotations. The VAE is not given any information about the ground truth z; however, we encode the notion that c is independent of y by conditioning the decoder on the label. We use the VAE inference network to assign groups to the training data, and we run ARM-CML using these learned groups. In Table 2, we see that ARM-CML in this setting outperforms ERM and is competitive with TTT, which as discussed earlier encodes a strong inductive bias for solving this task. Figure 5 visualizes samples from the VAE for different values of y and c.

Method	WC	Avg
ERM	74.3 ± 1.7	93.6 ± 0.4
TTT	$\mathbf{81.1 \pm 0.3}$	$\mathbf{95.4 \pm 0.1}$
ARM-CML	$\mathbf{81.7 \pm 0.3}$	$\mathbf{95.2 \pm 0.3}$

Table 2: Using learned groups, ARM-CML outperforms ERM and matches the performance of TTT on rotated MNIST. This result may be improved by techniques for learning more diverse groups for meta-training.

Figure 5: Visualizing VAE samples conditioned on different values of y (x axis) and c (y axis). The VAE learns to use c to represent rotations.

This result suggests that, when group information is not provided, a viable approach is to learn groups for ARM methods. Discovering disentangled factors of variation without supervision is, in the most general sense, an impossible problem [41]. However, when combined with meta-learning, the learned groups need not perfectly reflect the test time distributions; rather, the groups should cover many different distributions to allow for meta-training the model such that it can adapt to new test distributions. This advantage was noted by Hsu et al. [23], who show that even simple techniques such as overcomplete clustering can be effective for defining meta-training tasks. Incorporating techniques from this prior work is a promising direction for building on our results.

C.1 Qualitative analysis and observations

In Figure 6, we present an example of how ARM methods can improve test accuracy by adapting to specific users. We visualize a batch of 50 examples from a random FEMNIST test user, and we highlight an ambiguous example. An ERM trained model and an ARM-CML trained model, when only given a test batch size of 2 as shown by the black dashed box, incorrectly classify this example as "2". However, when given access to the entire batch of 50 images, which contain examples of class "2" and "a" from this user, the ARM-CML trained model successfully adapts its prediction to "a", which is the correct label. In general, we find that most examples of adaptation in FEMNIST occur for similarly ambiguous examples, e.g., "l" versus "I", though not all examples were interpretable.

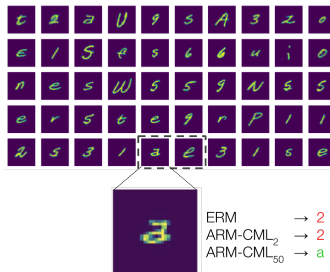

Figure 6: Visualizing one batch of 50 images from a FEMNIST test user. The ARM-CML model, using the entire batch, is able to successfully adapt to output the correct label "a" on the ambiguous example, shown enlarged, whereas other models incorrectly output "2".