Robustness Failures in CNNs

Bias Mitigation via Color Randomization and Loss Design

Neha Prabhu

February 8, 2026

Train-Test Split Failures:

The task wanted us to add colored bias to all digits with a random noise (of probability 95%). The document also hinted at diversifying the noise and not keeping the background a clean black.

I attempted to distribute both the train and test splits evenly with foreground strokes and background textures. This helped the model learn better, while the task required us to make it worse. Accuracy was spiking upto 45%. To mitigate this, I limited the training set to only comprise of images with biased foreground strokes, while the testing set contains a random distribution of foreground strokes and background texture. This increased bias very significantly and the model's performance dropped as we intended for it to.

The initial accuracy was justified by the concept of *Inductive Bias*. As epochs progress, it masters color but finds test accuracy to lack and thus learns shape automatically. The model is too smart, it needed to be dumber or the dataset construction needed to be harder.

Robust Model Creation Failures:

I have explored various algorithms to make a robust model to overcome the bias with the constraints as defined in the task document.

METHOD #1

I tried to use transforms while loading the datasets. This might work to a good extent, however it seemed to be on the edge of violation of the constraints specified – "without converting the image to grayscale and without changing the dataset (you still have the 95% bias)". This idea was discarded more on moral grounds.

```
train_transform_robust = transforms.Compose([
    transforms.ToTensor(),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5,
    hue=0.5),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
train_dataset_robust = CustomMNIST(train_imgs, train_labels,
    transform=train_transform_robust)
train_loader_robust = DataLoader(train_dataset_robust, batch_size=64,
    shuffle=True)
# test dataset is loaded as usual.
```

Transformation	Effect on the Model
ToTensor()	Converts images to PyTorch tensors and scales pixels to $[0, 1]$.
ColorJitter(hue=0.5)	Rotates the color wheel by up to 180°. It effectively "lies" to the model about what color the digit actually is.
Normalize(...)	Centers the data around 0.0 with a standard deviation of 1.0. This helps the optimizer (SGD) converge faster by keeping gradients stable.

The heart of this code is in the sect of `transforms.ColorJitter`. While standard augmentation might slightly tweak brightness, the parameters here are set to aggressive levels tuned to 0.5. Brightness, Contrast, Saturation bring out a change in the pixel values. This prevents the model from relying on specific shades or lighting conditions. Hue ensures that if an image was observed with 'color1' in the first, it is observed with any other color at least once during the training, motivating the model to change its learning strategy.

METHOD #2

Here, We explicitly define the model parameters and forward-pass function to fit to our bias scheme. This is as displayed below:

```
class ReweightedLoss(nn.Module):
    def __init__(self, outlier_weight=20.0):
        super().__init__()
        self.outlier_weight = outlier_weight
        self.ce = nn.CrossEntropyLoss(reduction='none') # Keep individual losses

    def forward(self, preds, targets, colors):
        raw_losses = self.ce(preds, targets)
        weights = torch.ones_like(raw_losses)
        with torch.no_grad():
            pred_labels = torch.argmax(preds, dim=1)
            wrong_preds = (pred_labels != targets)
            weights[wrong_preds] = self.outlier_weight
        return (raw_losses * weights).mean()
```

While the idea of penalizing the model even for the 5% outlier was novel, this assumed too much of a whitebox attack. I wanted to find something more general.

METHOD #3

We perform consistency regulation in this method by defining our own custom fit function. The model definition remains exactly identical to that of the *Lazy Model*. Our philosophy revolves around the fact that the model should be able to predict the same value even if the color of the input image changes. If this condition is not met, we penalize the model heavily and try again. The loss function thus accounts for both accuracy and consistency.

```
    for xb, yb in train_dl:
            preds_orig = model(xb)
            loss_ce = loss_func(preds_orig, yb)

            xb_permuted = xb[:, [1, 2, 0], :, :]
.
```

```
            preds_perm = model(xb_permuted)

            loss_consistency = F.mse_loss(preds_perm, preds_orig.detach())

            total_loss = loss_ce + (consistency_weight * loss_consistency)

            total_loss.backward()
            opt.step()
            opt.zero_grad()

            train_loss_accum += total_loss.item()
            train_correct += (torch.argmax(preds_orig, dim=1) ==
    yb).float().sum().item()
            train_total += yb.size(0)
```

However, the train and test accuracies remained stagnant at around 2.306 and 9.8 while performing this fit. Thus, the model had apparently resorted to guessing among the 10 classes at one point (which would rightly attribute to around 10% accuracy).

Since the model was too afraid to make a prediction and resorted to random guessing, we decide to let its mistakes slide and let it learn for a few epochs based on colour and then force it to use shape.

```
current_weight = 0.0 if epoch < warmup_epochs else max_consistency_weight
```

```
 Starting Robust Training with Warmup...
Epoch logs:  Epoch 1/15 [WARMUP (Bias Allowed)] | Train Acc:  77.95% | Test
Acc (Hard Set):  8.00%
Epoch 2/15 [ROBUST (Bias Penalized)] | Train Acc:  57.11% | Test Acc (Hard
Set):  10.62%
Epoch 3/15 [ROBUST (Bias Penalized)] | Train Acc:  28.32% | Test Acc (Hard
Set):  15.11%
Epoch 4/15 [ROBUST (Bias Penalized)] | Train Acc:  57.79% | Test Acc (Hard
Set):  16.37%
...
Epoch 15/15 [ROBUST (Bias Penalized)] | Train Acc:  10.90% | Test Acc (Hard
Set):  10.20%
```

However, it was observed that the stagnation followed no matter the number of warm up epochs we allowed, this method was thus discarded in search of a more fundamentally simpler method.
The model structure happened to have some fundamental flaws:

- 'loss_consistency' had a vanishing problem : the model found a strange comfort in outputting the standard normal distribution.
- The permutation of the color channels work only if all colors in the color_map were of the form [0,0,1], [0,1,0] or [1,0,0], but we have various combinations and so, it is not apt.

METHOD #4

```
def random_color_warp(imgs):
```

```
    device = imgs.device
    B, C, H, W = imgs.shape
    color_matrix = torch.rand(3, 3).to(device) + 0.2
    imgs_perm = imgs.permute(0, 2, 3, 1)
    imgs_warped = torch.matmul(imgs_perm, color_matrix)
    imgs_warped = imgs_warped / imgs_warped.max()
    return imgs_warped.permute(0, 3, 1, 2)
```

We now account for all combinations of RGB values, in all ratio-proportions - allowing dark green, cyan, yellow etc. To fix this, we need to be more aggressive. Instead of just swapping channels, we multiply the image by a Random 3x3 Matrix. This turns a "Red" digit into ANY random color (Pink, Brown, Cyan, Grey, etc.) on the fly. It preserves the Shape (pixel location) perfectly but destroys the Color information completely. This forces the model to learn: "The label is correct regardless of what color the pixels are.

What we missed here is the background pattern! All this was focusing heavily on the foreground stroke, so we make the necessary changes in the correct notebook!