Precog: Computer Vision

The Lazy Artist - Overcoming Bias

Neha Prabhu
2024101058

February 8, 2026

https://github.com/NehaP1706/Precog_CV_CNN.git

Abstract

This document presents insights into the process of addressing biases in datasets for the classification of MNIST handwritten digits. We inject noise and controlled bias into the training and testing datasets to intentionally mislead and degrade the performance of a simple CNN model. We compare the accuracies of the same model across different variants of the associated training (*fit*) function. Throughout the paper, we also reference the performance of our model on the unbiased MNIST dataset for the same classification task in order to draw meaningful parallels. We visualize the neurons in the convolutional layers of the model, along with the "ideal digit" representations learned at the output layer. Through confusion matrices, adversarial attacks, and other visualizations, we aim to develop a deeper understanding of various aspects of Convolutional Neural Networks.

Task 6 was not attempted due to time constraints; other tasks were completed to a satisfactory level.

Contents

1 Methodologies **3**
1.1 Biased Canvas . . . 3
1.2 The Cheater . . . 4
1.3 The Prober . . . 4
1.4 The Interrogation . . . 5
1.4.1 METHOD 1: Transformers . . . 5
1.4.2 METHOD 2: Forward Pass Edits . . . 5
1.4.3 METHOD 3: Consistency Loss . . . 6
1.4.4 METHOD 4: Randomized Color Wrap . . . 7
1.4.5 METHOD 5: Absolute Random Component . . . 7
1.4.6 METHOD 6: Reduced Random Component . . . 7
1.4.7 Summary . . . 8
1.5 The Intervention . . . 8
1.6 The Invisible Cloak . . . 9

2 Results & Analysis **9**
2.1 Per Layer Visualization . . . 9
2.1.1 Lazy Model . . . 10
2.1.2 Robust Model . . . 10
2.2 Ideal Digit Visualization . . . 10
2.2.1 Lazy Model . . . 11
2.2.2 Robust Model . . . 11
2.3 Confusion Matrices . . . 11
2.3.1 Lazy Model . . . 12
2.3.2 Robust Model . . . 13
2.4 Wrong Samples . . . 13
2.4.1 Lazy Model . . . 14
2.4.2 Robust Model . . . 15
2.5 Grad CAM Visualizations . . . 15
2.5.1 Lazy Model . . . 16
2.5.2 Robust Model . . . 17
2.6 Adversarial Attack . . . 17

3 Conclusion & Future Scope **18**
3.1 Concluson . . . 18
3.2 Future Scope . . . 18

4 References **19**

1 Methodologies

1.1 Biased Canvas

A *color map* is defined for all the 10 digits of interest for classification. This shall remain consistent throughout the paper and be referred to as the 'color-map'.

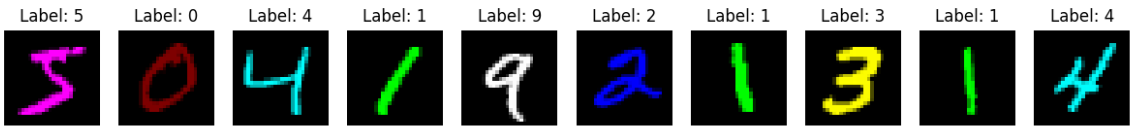
The algorithm to introduce bias into the MNIST dataset is as follows:

- **Training samples:**
  - With a probability of 95%, we allot the color as specified in 'color-map'.
  - With a probability of 5%, we allot a color distinct from the one specified in 'color-map'.
- **Testing samples:**
  - We allot a color distinct from the one specified in 'color-map'.
  - This is applied as a random distribution of background textures an foreground strokes.

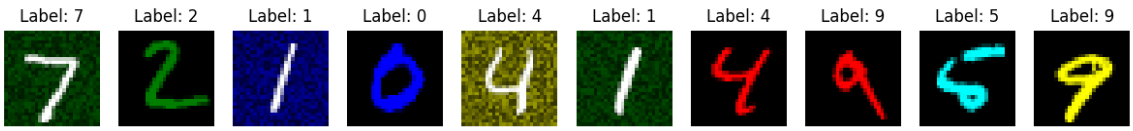
On an image of normalized pixel values, Foreground strokes and Background textures are implemented as follows:

- **Foreground Stroke:**
  - We color the digit pixels (White or bright grey) with values of its designated color according to 'color-map'.
  - This is achieved by simple element wise multiplication. As bright pixels tend to have an RGB value of 1, upon multiplication they yield the desired color (multiplier).
- **Testing samples:**
  - We generate a random noise factor and incorporate it into the chosen color's channel values.
  - We allow the white pixel values to remain unaffected, and incorporate the chosen color in varying brightness for the background pixels producing a *texture* as required. This is done by performing multiplication (as discussed earlier) with the inverted form of the pixel value.

A sample of the combined dataset as a result of this injecture is as follows:



(a) Training Set — Easy Bias



(b) Test Set — Hard Bias

Figure 1: Visualization of biased MNIST samples.

1.2 The Cheater

We define our simple model to be:

```
model = nn.Sequential(
    nn.Conv2d(3, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 16, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.Conv2d(16, 10, kernel_size=3, stride=2, padding=1),
    nn.ReLU(),
    nn.AdaptiveAvgPool2d(1),
    Lambda(lambda x: x.view(x.size(0), -1)),
)
```

We define a simple CNN Model using 'nn.Sequential' that represents the linear order of layers defined within. Features of the *lazy model* (as it shall be referred to here on) are discussed below.

The number of input features of the first layer is defined as 3 corresponding to the number of color channels (RGB). The number of output features are defined as 16, implying the use of 16 filters, a decent number for a small CNN. The model comprises of a single hidden layer with both input features and output features set to 16 for reduced complexity. The final output layer of the model has 10 output features to indicate the confidence (prediction logits) of the model classification. Standard activation functions like *ReLU* and *AvgPool* have been used between all layers.

The structure of the model is inspired (but heavily simplified for the present use-case) by the amazing visualizer CNN Explainer.

We use a standard fit function that minimizes the Cross Entropy Loss over a defined number of epochs with SGD as the optimizer. We define the hyperparameters such as learning rate, momentum etc. with values that worked out to be experimentally better and with reference to blogs cited in References.

1.3 The Prober

To complete this task, we produce visualizations of both the convolution layers and the final output layer. We plot the activations of the filters of all the convolution layers and bring out inferences about their features of focus. Additionally, the output layer is dissected to reveal the "Ideal digit" as perceived by the *lazy model*.

The neuron visualizer essentially performs the following iteratively for every filter and convolution layer in our implementation:

- We initialize a random image tensor of the same dimensions as that of the training samples.
- We set the model to be in evaluation and use the 'register_hook()' method to track activations and gradients that pass through intermediate layers in the model.
- We define a loss function to minimize the negative of activations and hence, maximize the activation content per filter.
- The initial image is normalized and rendered for output.

To visualize the perception of the "Ideal Digit", the main philosophy used is that the weights of the model remain frozen, the pixels of the image are tuned to cater to the classification of the model. The following computations are performed to derive this:

- We initialize a random image tensor of the same dimensions as that of the training samples.
- We track the gradients of the image pixel values over the weights of the model.

- We define a loss function to minimize the logits of pixels and hence, maximize the logits or confidence of model prediction for the given target label.
- To make the final output human-readable, we perform some regularization on the image pixels before every forward pass. This shifts the image around a few times, causing a single structure to appear instead of a confusing combination of several.
- To prevent the washout by outlier pixel values, it chooses bits between the 1st and 99th percentile for final rendering. The initial image is normalized and presented as output.

1.4 The Interrogation

The next natural task would be to improve the model's performance and that is exactly what we shall explore next. Under the constraints of the task, we are expected to get creative with the fit functions while keeping the dataset and model architecture identical to that of our *Lazy Model*. We approach this task iteratively with several ideas.

1.4.1 METHOD 1: Transformers

The philosophy of the first method stems from the usage of the transformers library to perform data augmentation at the time of data loading. This is a novel method as it would eliminate the bias at the root of the learning process and leads to no generalization issues.

```
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)
```

This method involves converting the image to a tensor of pixel values, manipulating the visual features of the image and normalizing the same before feeding it to the model. This essentially renders all colors to change to other random colors before the model can attempt to learn "color". While this would be a natural path to take, it stands close to violating the task constraints and thus, was not taken further after reading.

1.4.2 METHOD 2: Forward Pass Edits

We now redefine the model definition with custom parameters and forward pass functions over the standard ones defined for the *Lazy Model*. The updates involve an outlier weight indicator that penalized the model severely when the outliers were not assigned their true values.

This method assumes a rather white-box access to the model and does not follow a standard. This method does not do well under generalization as is apparent by the specificity to model type.

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

    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()
```

1.4.3 METHOD 3: Consistency Loss

We perform consistency regulation in this method by defining our own custom fit function. 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.

1.4.4 METHOD 4: Randomized Color Wrap

We further realize that permuting values of [R, G, B] may not account for all possible colors defined in the 'color map' that may be present in the testing sample. This may cause the model to stagger significantly and hence, we propose forming a random color warp on the training image before performing a forward pass in the fit function.

```
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)
```

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. However, a key aspect that was missed in this implementation was the presence of the background texture. It was not randomized and handled.

1.4.5 METHOD 5: Absolute Random Component

To bridge the gap that the previous model faced, we randomize both the foreground and background components of the training sample digit independently. The motivation is similar to that of the previous few models, the model is made to forget about correlations with color and focus on shape.

This function was called in the fit function for every training sample right before the model could predict on the same.

```
def randomize_components(imgs):
    device = imgs.device
    B, C, H, W = imgs.shape
    mask = (imgs.mean(dim=1, keepdim=True) > 0.1).float()
    fg_color = torch.rand(B, 3, 1, 1).to(device)
    bg_color = torch.rand(B, 3, 1, 1).to(device)
    imgs_aug = (mask * fg_color) + ((1 - mask) * bg_color)
    return imgs_aug
```

However, it was observed that the accuracy of the model although respectable was struggling at 68%. This seemed to be on the edge, because based on the task description we needed a model with at least 70% test accuracy.

1.4.6 METHOD 6: Reduced Random Component

Thus, we allow a small amount of leisure on the model and randomize the image components with a probability of 95%. This pushed the accuracy upto 75% for certain random seed values.

```
def randomize_components(imgs, prob=0.95):
    device = imgs.device
    B, C, H, W = imgs.shape
    n = np.random.rand()
    if n > 0.95:
      return imgs
    mask = (imgs.mean(dim=1, keepdim=True) > 0.1).float()
    fg_color = torch.rand(B, 3, 1, 1).to(device)
    bg_color = torch.rand(B, 3, 1, 1).to(device)
    imgs_aug = (mask * fg_color) + ((1 - mask) * bg_color)
    return imgs_aug
```

1.4.7 Summary

To provide a gist of all the trials, consider the following tabulation:

Table 1: Comparison of Bias-Mitigation Strategies

Method	Pros	Cons
Method #1	Encourages the model to treat color as noise and focus on shape. Simple to implement using standard augmentation tools.	Borderline violation of task constraints. May not remove bias entirely if background patterns remain consistent. Still allows the model to exploit residual color cues.
Method #2	Directly targets misclassified "outlier" samples. Conceptually aligns with emphasizing rare unbiased signals.	Assumes white-box knowledge of model errors. Training became unstable; accuracy stagnated near random guessing. Model developed degenerate output behavior.
Method #3	Allows model to learn basic structure before enforcing robustness. Attempts gradual transition from biased to unbiased learning.	Performance plateaued despite warm-up tuning. Model reverted to guessing; bias still dominated learning.
Method #4	Destroys color information completely while preserving shape. Covers all RGB combinations, not just channel swaps.	Did not address background texture bias. Aggressive transforms may distort useful low-level features.
Method #5	Completely removes color–label correlation during training.	May over-randomize and reduce learning of stable low-level cues. Can slow convergence and hurt clean-data performance.
Method #6	Breaks correlation between digit shape and color/texture. Forces the model to rely on structural features rather than color bias.	Random recoloring can remove useful intensity cues. No guarantee of robustness to unseen bias patterns.

We name this model the 'Robust Model'. It shall be referred to as with the same name.

1.5 The Intervention

This task is key to proving the laziness of our model mathematically. We leverage the mentioned 'pytorch-gradcam' algorithm properties to do exactly this! The GradCAM class below recognizes what features the model recognized and to what extent the pattern influenced its decision.

Since 'pytorch' doesn't save the intermediate outputs, we use 'register_forward_hooks()' to track activations and 'register_full_backward_hook()' to save the gradients which is crucial for the heat map over-lay later.

In the main method of our custom class we perform the crux of the 'pytorch-gradcam': - We pick the class we want to explain the gradient of (with a relevant feedback), pass it through the model and calculate its gradient activations. - We calculate an average-pool for the gradient over all filters that indicates its "importance weight". This is followed by combining all feature maps where the significant ones are amplified and the irrelevant ones are discarded. - Since we are only concerned about positive effect, we discard the negative values using the activation function - 'ReLU()'.

This is presented as follows:

```
    def __call__(self, x, class_idx=None):
        self.model.eval()
        self.model.zero_grad()

        output = self.model(x)

        if class_idx is None:
            class_idx = torch.argmax(output, dim=1).item()

        score = output[0, class_idx]
        score.backward()

        grads = self.gradients
        fmap = self.activations
        weights = torch.mean(grads, dim=(2, 3), keepdim=True)

        cam = torch.sum(weights * fmap, dim=1, keepdim=True)
        cam = F.relu(cam)
        cam = cam - cam.min()
        cam = cam / (cam.max() + 1e-7)

        cam = F.interpolate(cam, size=x.shape[2:], mode='bilinear',
    align_corners=False)

        return cam.detach().cpu().numpy()[0, 0], class_idx
```

1.6 The Invisible Cloak

In this task, we are to perform an adversarial target attack on our model to evaluate its robustness. This attack is not a backdoor attack but rather a form of noise injection that is invisible to the naked eye but confidently deviant to the model classification logits.

We define two helper functions to perform this attack:

- **The Attack:**
  - We start with a clean image and slightly nudge its pixels until the model classifies it with the wrong label (as needed).
  - We optimize the noise and not the model, the loss function being the difference between the model's current prediction (mostly the true value) and the targeted wrong label.
  - We apply FGSM iteratively reducing the loss while clamping the values so that the change (a range in ϵ) remains invisible to the naked eye.
- **The Evaluation:**
  - It iterates through various values of ϵ from 0.0 to 0.5, and finds the value for which the model mis-predicts with a high confidence score ($> 90\%$).

2 Results & Analysis

2.1 Per Layer Visualization

With the method described in this Section, we derive the following plots for the models:

2.1.1 Lazy Model

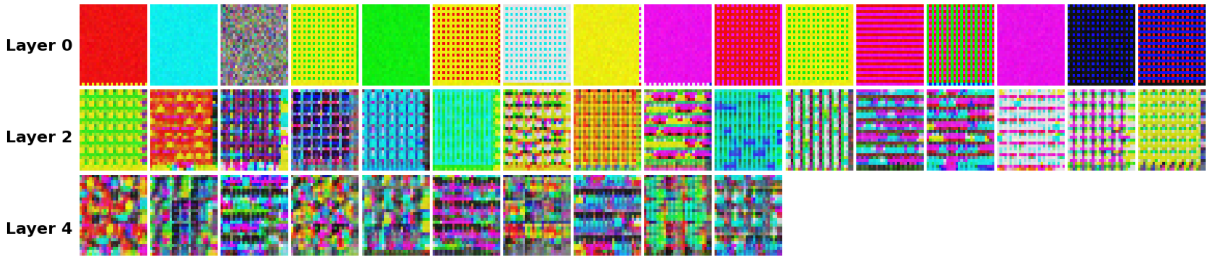


Figure 2: Feature Visualization of Lazy Model

As expected, the effect of color on the model's learning is extremely high. We infer:

- Layer 0 is completely based on color or a grid-like combination of two colors.
- Layer 2 uses combinations of colors but retains a grid-like search pattern and doesn't develop any intuition on structures of numbers. Light hints on the curves seem to appear.
- Layer 4 has a stronger mix of colors with more structure but still heavily non-indicative of the knowledge it should've ideally learnt.

2.1.2 Robust Model

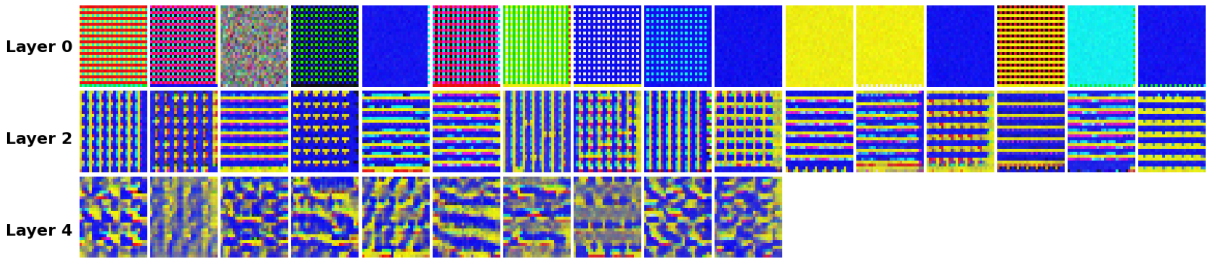


Figure 3: Feature Visualization of Robust Model

While the filters in the first layer of the Robust Model corresponds to that of the Lazy Model with its bias in considering colour for features exclusively, the trend does not hold past that layer. In contrast to the filters of the *Lazy Model*, the final layer sees no correlation with color at all! It is majorly based on texture, shape and curves, as we needed!

2.2 Ideal Digit Visualization

With the method described in this Section, we derive the following plots for the models:

Consider our color map as displayed in this Section.

2.2.1 Lazy Model

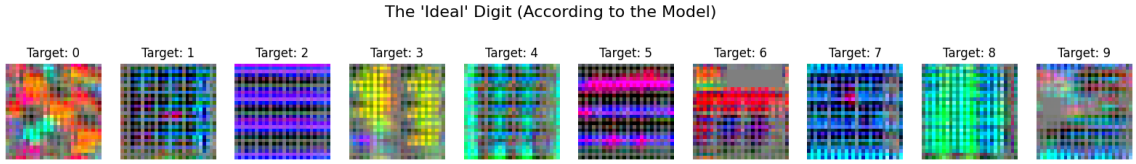


Figure 4: Ideal digit perceived by Lazy Model

For digits 0, 2, 3, 4, 5, 7 and 8 - their plots have strong presence of their characteristic color (red, blue, yellow, cyan, pink, dark_blue and green). This is very indicative of the model's laziness. Yet, it still has appeared to learn the structure of certain digits near-concretely (3 and 5).

2.2.2 Robust Model

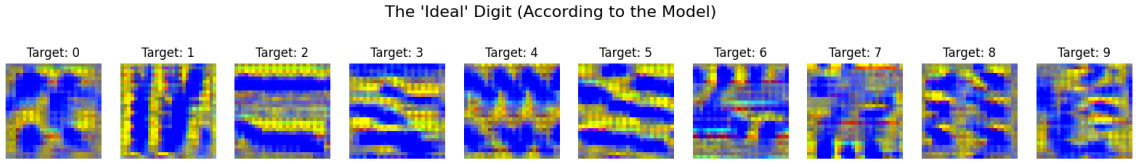


Figure 5: Ideal digit perceived by Robust Model

This is immensely interesting for several reasons:

- 0: We can follow the blue regions of the image to form a loop prevalent in the digit 0, this indicates a near success.
- 1: We can find angled lines in both yellow and blue, all attributing to the several possible positions of 1 in the grid space.
- 2, 4, 7, 9 seem to appear to be too hazy for recognition even with this model. If one follows the blue/yellow swirls in the image for 3 one can see the potential for 3 being present. Similar cases follow for 5, 6 and 8.

2.3 Confusion Matrices

Using the code snippet below, we generate the confusion matrices for both the models.

```
model.eval()
all_preds = []
all_labels = []

with torch.no_grad():
    for xb, yb in valid_dl:
        preds = torch.argmax(model(xb), dim=1)
        all_preds.extend(preds.cpu().numpy())
        all_labels.extend(yb.cpu().numpy())

cm = confusion_matrix(all_labels, all_preds)
```

```
12
13 plt.figure(figsize=(10, 8))
14 sns.heatmap(cm, annot=True, fmt='d', cmap='rocket', cbar=True)
15
16 plt.title("Confusion Matrix (Test Set)")
17 plt.xlabel("Predicted Label")
18 plt.ylabel("True Label")
19 plt.show()
```

2.3.1 Lazy Model

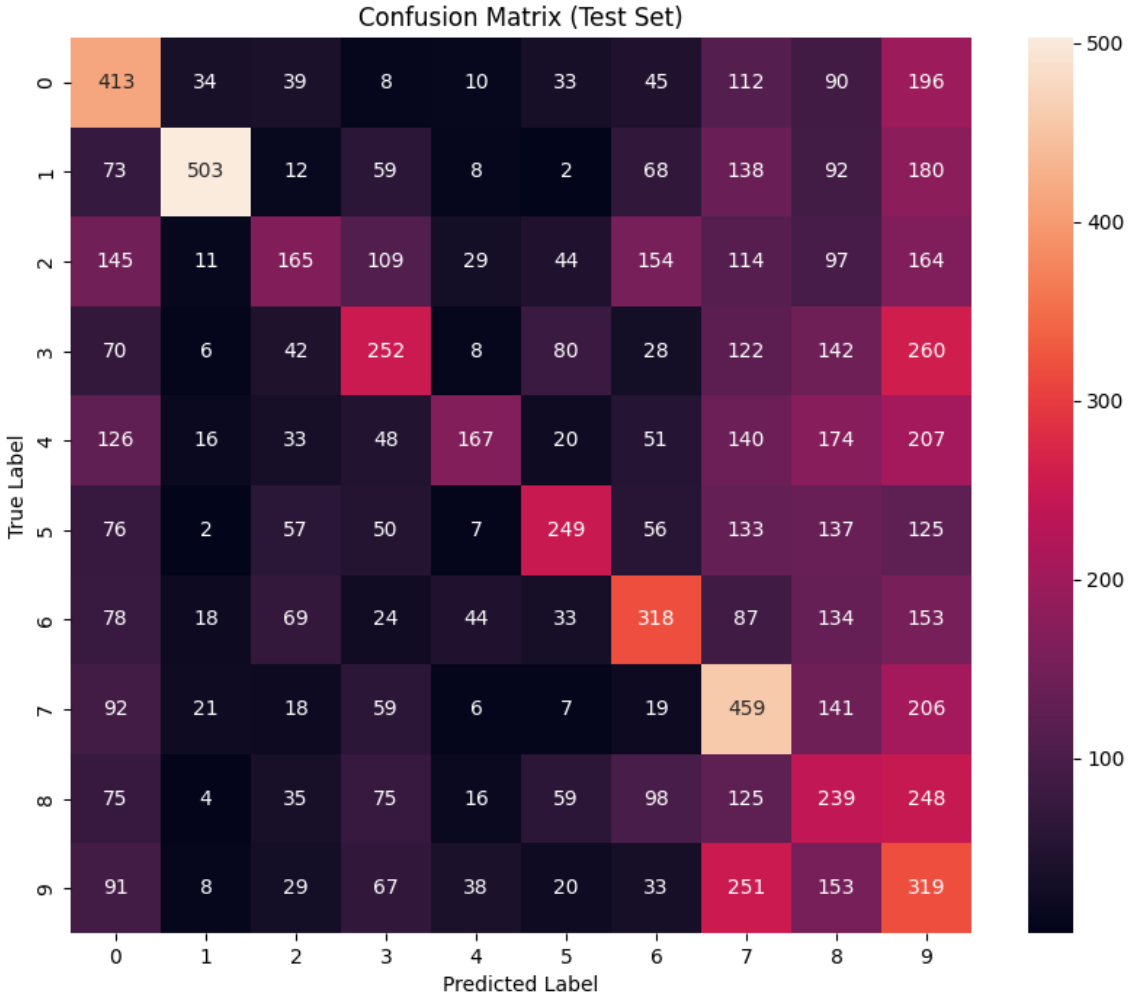


Figure 6: Confusion matrix derived from Lazy Model

The model performance is nearly up-to-the mark as indicated by the diagonal nature of the confusion matrix. However, it is significantly confused with prediction digits like 8 and 9, misplacing many of them sparsely withing buckets allocated for other digits. This is not ideal and can be improved upon.

2.3.2 Robust Model

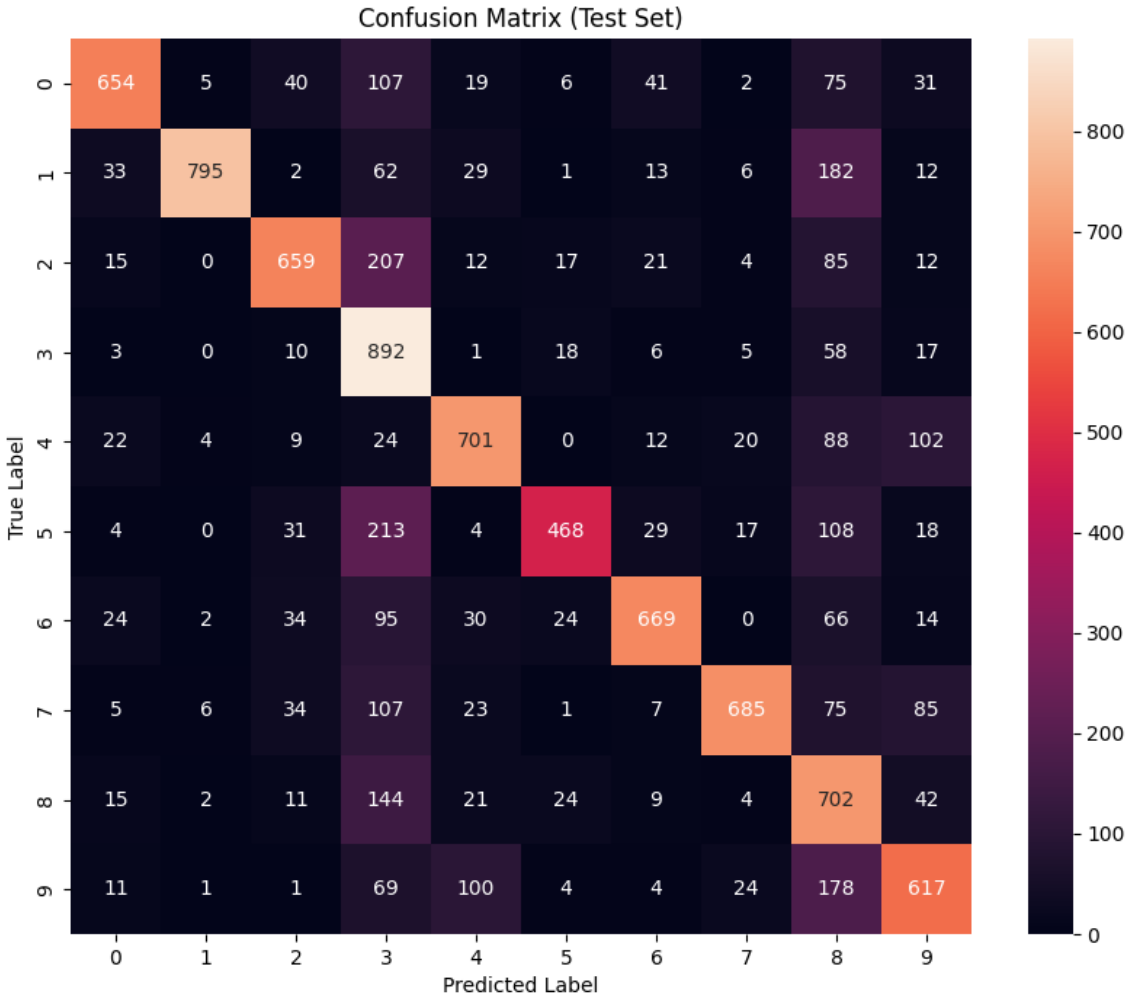


Figure 7: Confusion matrix derived from Robust Model

As we stated previously, the confusion in classifying digits 8 and 9 can be taken care of with further optimization. The Robust Model has executed just that with splendid accuracy. This proves our custom fit function was highly appropriate and the Lazy Model's tactics could be overcome.

2.4 Wrong Samples

We consider a random set of 3 samples that were predicted incorrectly by the model and present inferences on them in this section.

Consider our color map:

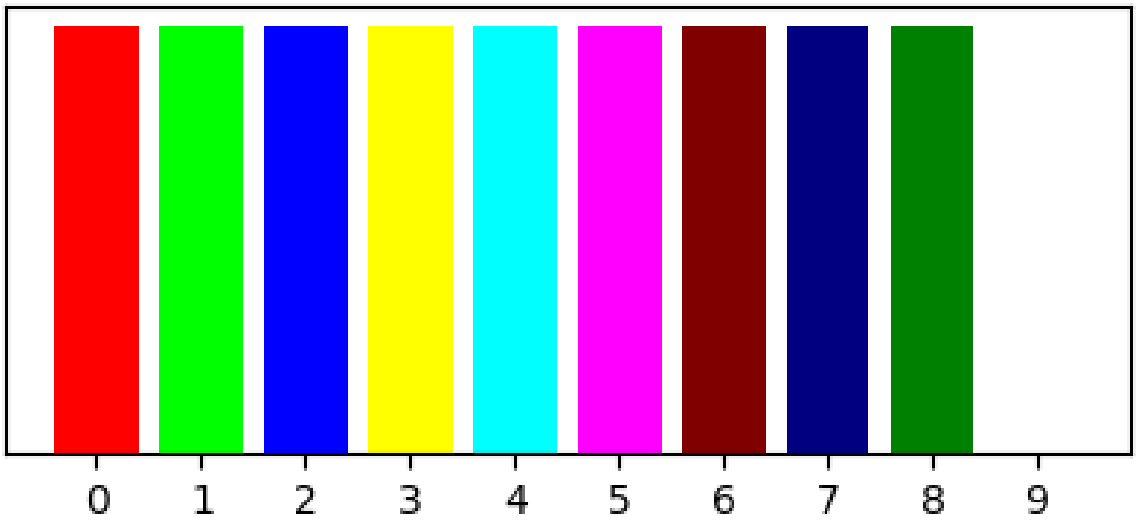


Figure 8: Wrong Predictions sampled from Lazy Model

2.4.1 Lazy Model

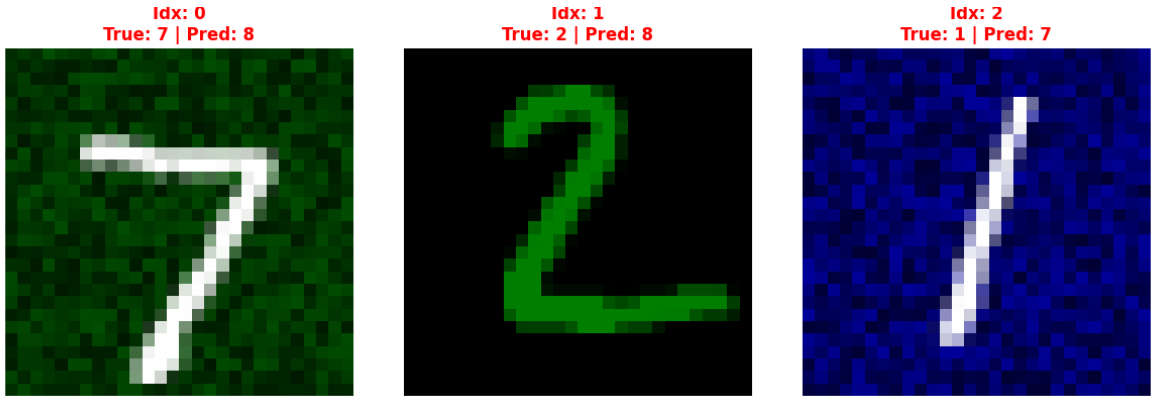


Figure 9: Wrong Predictions sampled from Lazy Model

As we infer,

- 7 and 2 have elements of green as a background texture and foreground stroke respectively. This color maps to 8 which explains the incorrect prediction value and brings out the inherent confusion.
- Similarly, 1 has a background texture of blue which was designated to 7. This matches the expectation of the model to perform sub-par.

Thus, the laziness of the Lazy Model is in full show and the tactic is clear. We shall contrast this with the human-explainable nature of the errors produced by the Robust Model. We consider these errors to be genuinely difficult to resolve, even for a human at times.

2.4.2 Robust Model

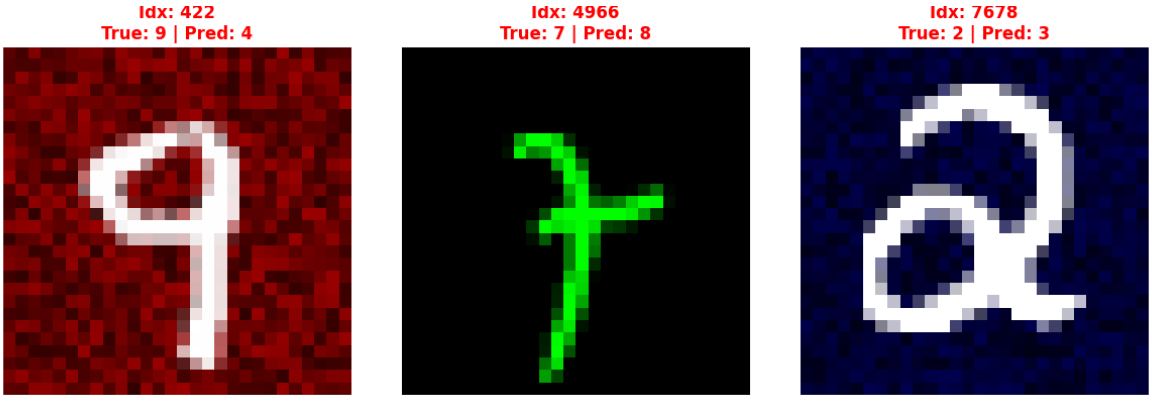


Figure 10: Wrong Predictions sampled from Robust Model

As visible above, the errors in prediction are completely unrelated to the initial color maps!

- 9 and 4 can often be confused with each other even as humans due to the smoothness of the curve that forms at the top.
- 7 and 8, while usually not to be confused each other, the model may seem to focus on the intersection on the lines and conclude on an 8 when a 7 is rendered the way it is above.
- Similar logic may be used for the misprediction of the handwritten 2 above.

2.5 Grad CAM Visualizations

With the method described in this Section and the code snippet below, we derive the following plots for the models:

```
for i in range(10):
      img = images_to_show[i]
      device = next(model.parameters()).device
      img_input = img.unsqueeze(0).to(device).requires_grad_(True)

      heatmap, pred_idx = grad_cam(img_input)

      img_disp = img.permute(1, 2, 0).cpu().numpy()

      ax_orig.imshow(img_disp)
      ax_orig.axis('off')

      ax_heat = axes[row_offset + 1, col_idx]
      ax_heat.imshow(img_disp)
      ax_heat.imshow(heatmap, cmap='jet', alpha=0.5)
      ax_heat.set_title(f"Grad-CAM {i}", fontsize=10)
      ax_heat.axis('off')
```

2.5.1 Lazy Model

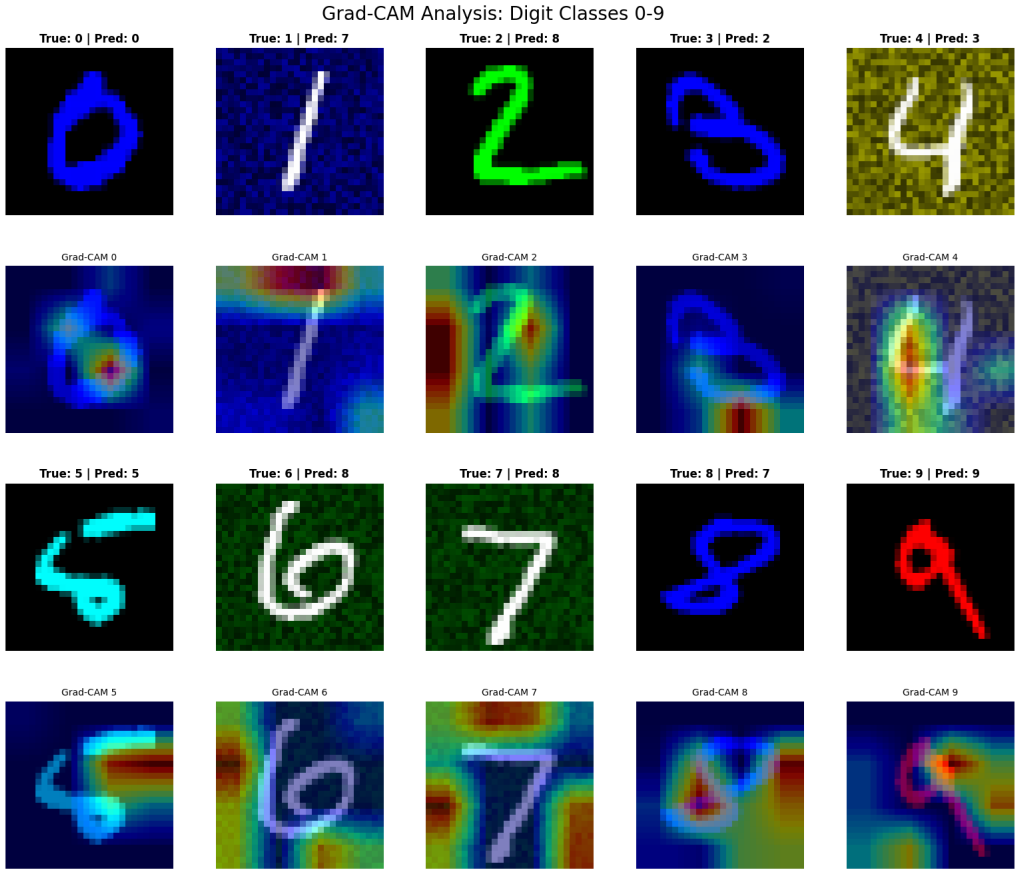


Figure 11: Grad CAM of Lazy Model on Testing samples

As evident from the custom GradCAM for certain digits like 1, 2, 5, 6, 7 and 9 the model does not seem to concentrate on the shape of the underlying image but focuses on the background. It even focuses on certain aspects of the shape but not all of it. Examples of the latter include 0, 3 and 4 - while the model does concentrate on a part of the shape one could argue that it could not possibly be enough to predict on the digit class.

This set of GradCAM results has been produced on the testing samples fed to the Lazy Model, evident by the mixture of foreground strokes and background textures among the samples. Similar GradCAM results when produced on the training samples produce more positive results. The Model does not appear to focus on the background as exclusively as it does for the testing samples. This is further proof of the the mathematical validity of our claim.

Consider the GradCAM results on the Robust Model as follows to draw parallels.

2.5.2 Robust Model

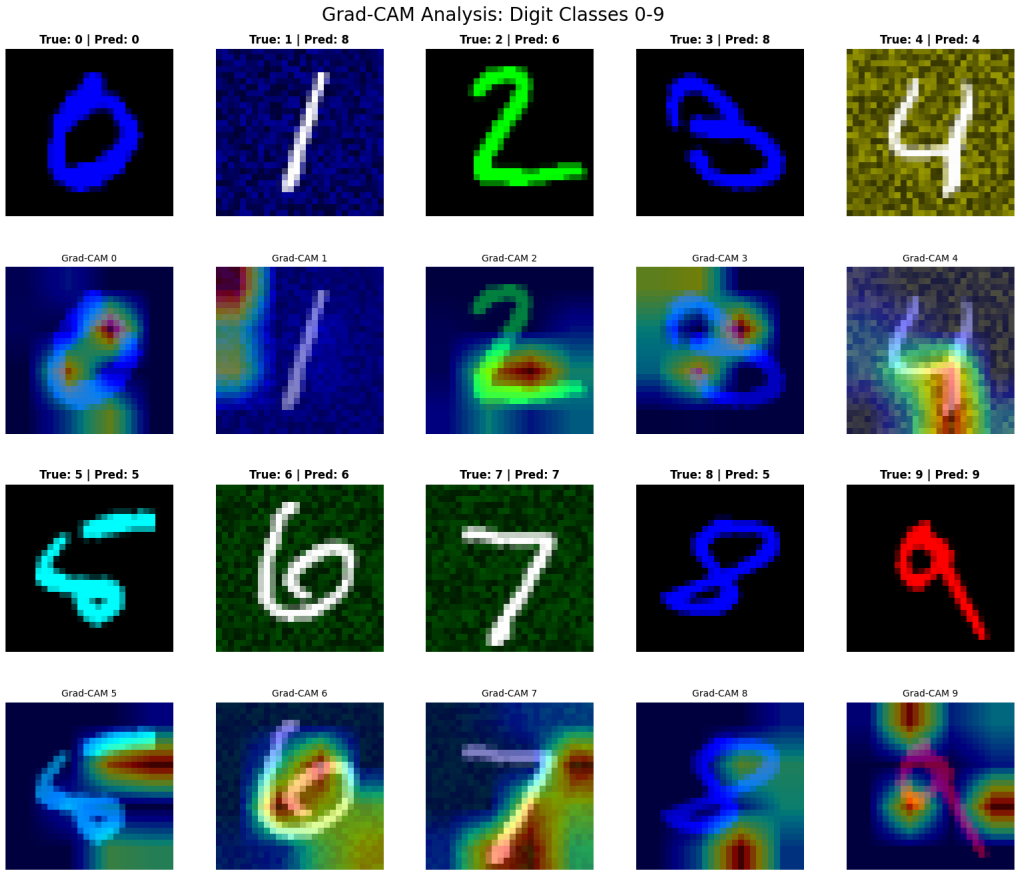


Figure 12: Grad CAM of Robust Model on Testing samples

The results for the GradCAM when applied on testing samples evaluated by the Robust Model do not show strong correlation with shape, structure and curvature of the digits. This is concerning as the test accuracy is decently high overall.

For digits like 0, 2, 4 and 6 the model seems to focus on distinguishing curves and intersections of the digits. However, when it comes to the others, there is still a significant proportion focusing on either the background or part of a seemingly unrelated curve.

This is indicative of the fact that the Robust Model can be improved immensely.

2.6 Adversarial Attack

The higher the ϵ value, the more robust and secure the model is considered. We have considered ϵ values ranging from 0.0 to 0.5 and our model cracks at $\epsilon = 0.04$. This is indicative of the highly brittle nature of our Robust Model. This is not ideal and can be improved.

Figure 13: Adversarial Attack Results

3 Conclusion & Future Scope

3.1 Concluson

This work set out to explore how a convolutional neural network behaves under biased data distributions and whether simple training-time tweaks could push the model toward learning more meaningful, shape-based representations.

We successfully induced a strong color–label shortcut that caused the Lazy Model to rely heavily on a color map rather than digit structure and shape. This behavior was confirmed not only through accuracy trends but also through neuron visualization, "ideal digit" generation, confusion matrices, Grad-CAM analysis, and error inspection.

The Lazy Model demonstrated high dependence on background textures and foreground color cues. Additionally, wrong predictions were directly traceable to the color-map bias. This validated the hypothesis that standard CNNs, when exposed to biased training distributions, will exploit the easiest signal available, even when that signal is meaningless.

This work explored a structured approach to improving the Lazy Model. In particular, the probabilistic method entailing a custom fit function struck a balance between invariance pressure and training stability, improving test performance to the required level while encouraging the model to encode more structural cues.

However, analysis of Grad-CAM on the Robust Model revealed that although color alliance was no longer present, attention was still not perfectly aligned with digit structure and shape. Due to the brittle nature of the Robust Model towards the Adversarial Attack, we conclude that robustness to distribution bias does not directly translate to robustness against input perturbations.

Thus, while bias-removal through training-time augmentation proved partially successful, it does not solve deeper robustness limitations of CNNs.

3.2 Future Scope

Self-supervised algorithms can be employed in conjunction with the current probabilistic and randomized approaches to explore different directions in improving the simple CNN model. Training period can be made stricter with shifts, rotations and other actions on samples forcing the model to recognize shape. architectural modifications such as shape-biased CNNs, anti-aliasing filters, or hybrid CNN–ViT models may improve the performance significantly.

4 References

References

[1] "Visualizing Convolutional Features in 40 Lines of Code." *Medium*. https://medium.com/data-science/how-to-visualize-convolutional-features-in-40-lines-of-code-70b7d87b0030

[2] Polo Club of Data Science. "CNN Explainer." https://poloclub.github.io/cnn-explainer/

[3] TensorFlow Lucid: Neural Network Feature Visualization Library. https://github.com/tensorflow/lucid

[4] Lucent: PyTorch Feature Visualization Library. https://github.com/greentfrapp/lucent

[5] Nan Bhaskhar. "Intermediate Activations — the Forward Hook." https://web.stanford.edu/~nanbhas/blog/forward-hooks-pytorch/

[6] Keras Documentation. "Grad-CAM Class Activation Visualization." https://keras.io/examples/vision/grad_cam/

[7] Stepan Ulyanin. "Implementing Grad-CAM in PyTorch." *Medium*. https://medium.com/@stepanulyanin/implementing-grad-cam-in-pytorch-ea0937c31e82

[8] LearnOpenCV. "Introduction to Grad-CAM." https://learnopencv.com/intro-to-gradcam/

[9] ImageNet-trained CNNs are biased towards texture; increasing shape bias improves accuracy and robustness https://arxiv.org/abs/1811.12231

[10] MNIST-C: A Robustness Benchmark for Computer Vision https://arxiv.org/pdf/1906.02337

[11] "Domain Randomization for Neural Networks." *Medium*. https://medium.com/data-science/domain-randomization-c7942ed66583

[12] Learn PyTorch for deep learning in a day. Literally. *Daniel Bourke, YouTube* https://www.youtube.com/watch?v=Z_ikDlimN6A