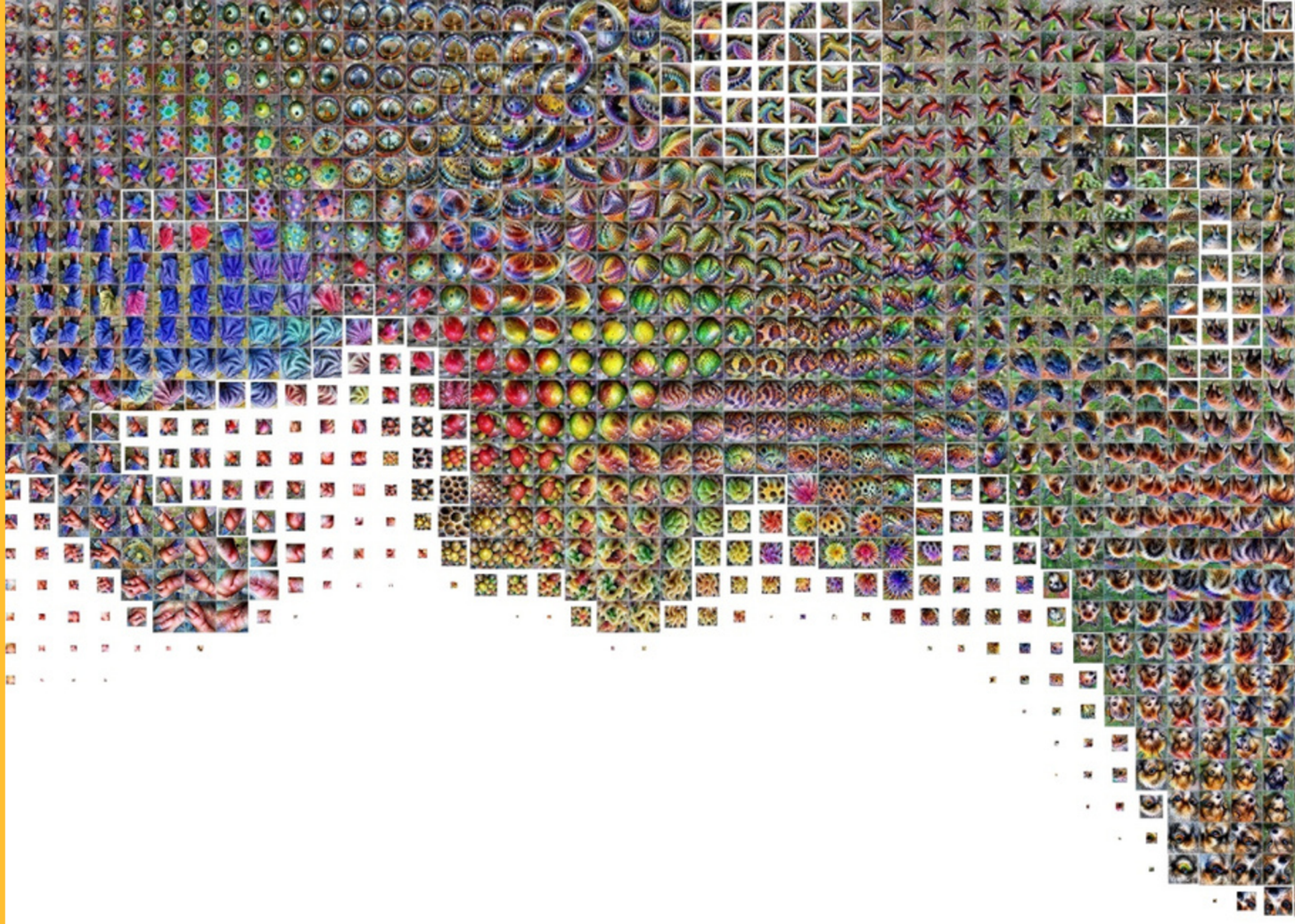


Computer Vision

Akshat Gupta

Tasks done by me

01 The Biased Canvas

02 The Cheater

03 The Prober

04 The Interrogation

05 The Intervention

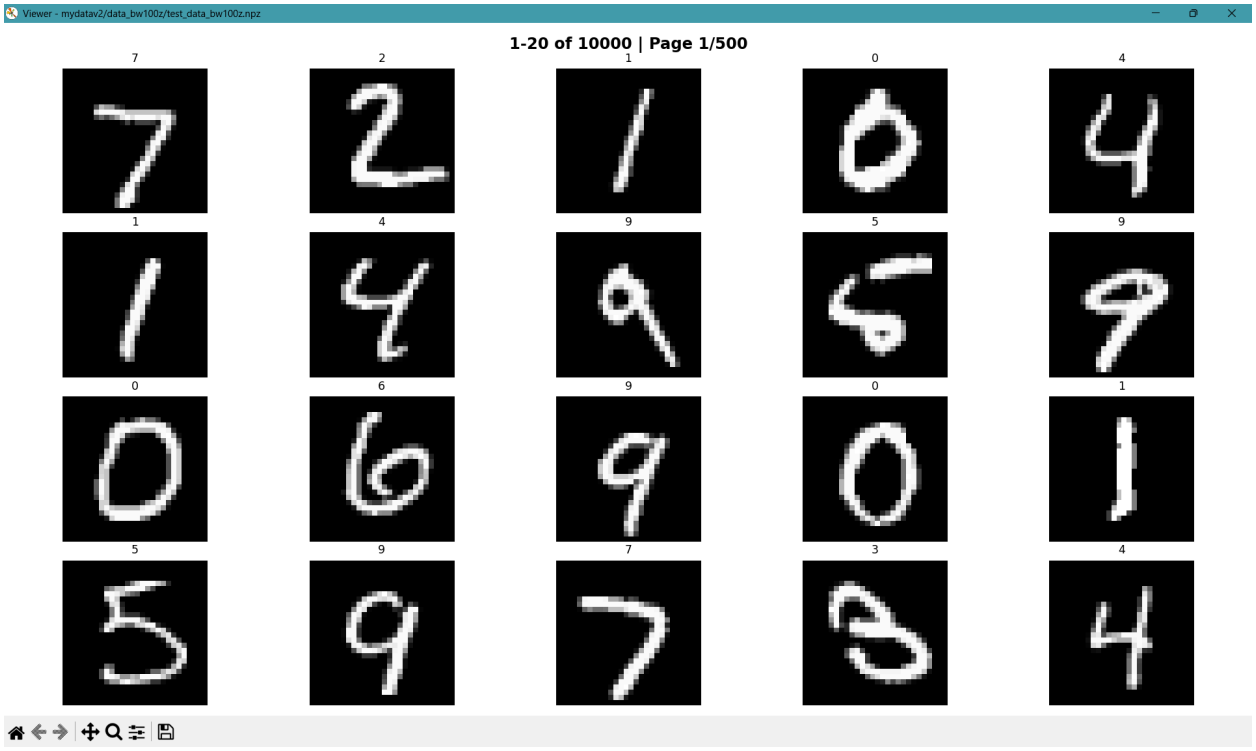
06 The Invisible Cloak

07 The Decomposition

The Biased Canvas

By far the most popular dataset for digits is the MNIST dataset by Yann LeCun, Corinna Cortes, and Christopher Burges(1). It consists of 60000 training and 10000 testing images of black and white handwritten digits of 28x28 resolution. The dataset has several variations, coloured MNIST, fashion MNIST, earliest by US Postal Department who developed it to train models to classify the legible handwriting on postcards. I think originally the dataset was available at http://yann.lecun.com/exdb/mnist/ but now it is not there anymore. Thankfully various organisations (like kaggle,tensorflow,huggingface,microsoft) and individuals maintain versions and clones/mirrors of the dataset. I downloaded mine from this public github repo - https://github.com/cvdfoundation/mnist?tab=readme-ov-file.

There were 4 files, test-data,test-label,train-data,train-label in .idx3-ubyte format. I made a script (convert.py) to convert this into .npz format for easy handling,manipulation and loading. I also made another script (visualize.py) to view the dataset.



I had successfully gone to the market and procured my canvas, albeit for free. Now I had to paint it. I wrote a script (colorise.py) this script was changed multiple times by and I have attached only the final version. This script has made to help me paint the canvas as easily as possible in whichever way I want. In a dictionary I provide the dominant color for each digit.

“I automate cause I want to remain lazy” - Every engineer

A painter usually feels creative and paints as he likes, despite having metadata.txt generated for each dataset, I wanted a method by which by just reading the name of the dataset I knew its properties. So I made a naming convention which would help me systematically generate datasets of varying difficulties.

RG - this meant dominant colour for 0 is Red and for 1 is Green, other digits pairs like 2-3, 4-5 .. 8-9 has their own dominant colours

GR - this meant dominant colour biases were reversed. 0 is Green and 1 is Red. For any pair say 6-7 (brainrot ref lol) if 6 was orange and 7 purple, then 6 would be purple and 7 orange.

BW - this meant black background and white digits.

Tier	Noise Probability	Noise Range	Stroke Probability	Stroke Range	Stroke Thickness
e (Easy)	0.7	15, 25	0.5	(1, 3)	(1, 2)
m (Medium)	0.75	17.5, 27.5	0.55	(1, 3)	(1.25, 2)
h (Hard)	0.8	20, 30	0.6	(2, 4)	(1.5, 2.5)
vh (Very Hard)	0.85	22.5, 32.5	0.65	(2, 4)	(1.75, 2.75)
z (Zilch)	0	NA	0	NA	NA

So gr_95z meant green 0 and red 1, 95% of the time, and difficulty was z (zilch) meaning no background noise and no strokes.

Noise Methodology -

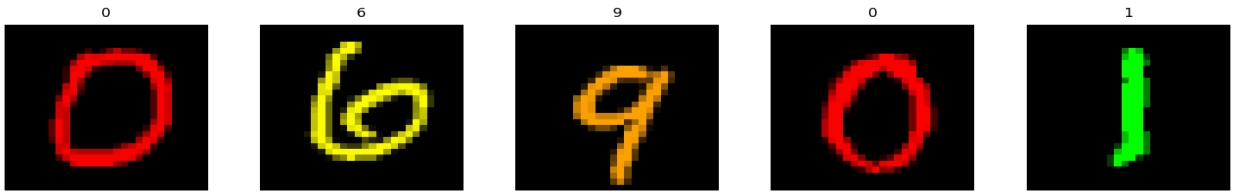
1 With probability noise_prob (70%), noise is applied
2 A random noise level is selected from noise_range (15-25)
3 Gaussian noise is generated using np.random.normal(0, level, result.shape) - mean=0, std deviation=level

4 The noise is added to each pixel's RGB values
5 Results are clipped to valid range [0, 255] to prevent overflow

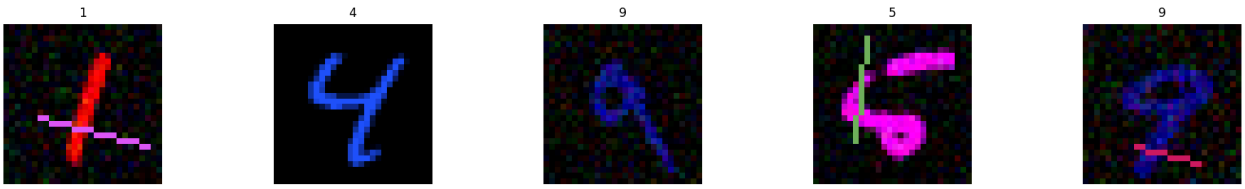
Stroke Methodology -

1 With probability 50%, random strokes are applied to the image.
2 A random number of strokes is selected uniformly between 1 and 3.
3 For each stroke, two random points within the image bounds are sampled, along with a completely random RGB color (0–255 per channel).
4 A random stroke thickness (1–2 pixels) is chosen, and a line is drawn between the two points using cv2.line().

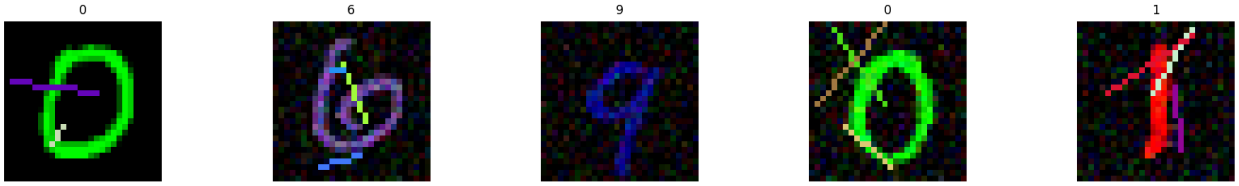
RG_95Z



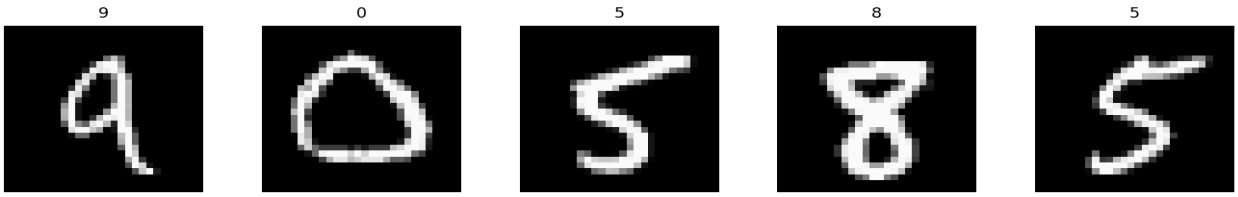
GR_95M



GR_95H



BW_100Z



This is how metadata.txt looks for rg_95z

```
================================================
DATASET METADATA
================================================

COLOR MAPPING:
  Mode: NORMAL

  Digit 0: RGB(np.uint8(255), np.uint8(0), np.uint8(0))
  Digit 1: RGB(np.uint8(0), np.uint8(255), np.uint8(0))
  Digit 2: RGB(np.uint8(0), np.uint8(255), np.uint8(255))
  Digit 3: RGB(np.uint8(255), np.uint8(192), np.uint8(203))
  Digit 4: RGB(np.uint8(255), np.uint8(0), np.uint8(255))
  Digit 5: RGB(np.uint8(31), np.uint8(81), np.uint8(255))
  Digit 6: RGB(np.uint8(255), np.uint8(255), np.uint8(0))
  Digit 7: RGB(np.uint8(120), np.uint8(81), np.uint8(169))
  Digit 8: RGB(np.uint8(18), np.uint8(10), np.uint8(143))
  Digit 9: RGB(np.uint8(255), np.uint8(165), np.uint8(0))

PARAMETERS:
  ALL_WHITE: False
  REVERSE: False
  BIAS: 0.95
  NOISE_PROB: 0
  NOISE_RANGE: (15, 25)
  STROKE_PROB: 0
  STROKE_RANGE: (1, 3)
  STROKE_THICK: (1, 2)
  PROGRESS: 1000
```

I generated the following datasets - bw95z, bw100z, gr95e, gr95h, gr95m, gr95vh, gr95z, gr100z, rg95z, rg100z. Each has metadata.txt,train_data.npz,test_data.npz (labels are inbuilt).I chose .npz because it efficiently stores multiple NumPy arrays in a single compressed file, making dataset loading fast, simple, and memory-efficient.

I have made the latest iteration of my dataset opensource on kaggle and it can be found here https://www.kaggle.com/datasets/akshatatkaggle/cmnistneo1

These are the old iterations (most recent first) (not recommended to use)

1 https://www.kaggle.com/datasets/akshatatkaggle/cmnist-v2
2 https://www.kaggle.com/datasets/akshatatkaggle/akshat-cmnist-dataset
3 https://www.kaggle.com/datasets/akshatatkaggle/mnist-coloured-dataset
4 https://www.kaggle.com/datasets/akshatatkaggle/task1-v1
5 https://www.kaggle.com/datasets/akshatatkaggle/mnist-v0

The Cheater

The Willing Accomplice

After preparing my canvas so meticulously, I now needed a painter. Not a very clever one just competent enough to reveal its own biases when tempted. I was not looking for robustness, fairness, or interpretability at this stage. I wanted something that would *take the bait*.

So I built a simple convolutional neural network, nothing fancy, no tricks, no moral compass.

The model consists of three convolutional layers followed by two fully connected layers. The input is a 3×28×28 RGB image, exactly matching the colored MNIST images I had generated. Each convolution is followed by a ReLU activation and max-pooling, progressively reducing spatial dimensions while increasing feature depth.

The architecture looks roughly like this:

- Conv layer: 3 → 32 channels
- Conv layer: 32 → 64 channels
- Conv layer: 64 → 64 channels
- Fully connected layer: 576 → 128
- Output layer: 128 → 10

Between the final hidden layer and the classifier, I added dropout nominally for regularization though at a rate so small (0.02) that it mostly serves as a decorative element rather than a meaningful constraint.

Adam was chosen as the optimizer. I used a learning rate of 0.001, which is absurdly high by any sensible deep learning standard. Normally this would be a recipe for divergence, instability, or silent failure. But Colored MNIST is forgiving. When the shortcut is strong enough, optimization sins are quickly absolved.

The model was trained for 30 epochs with a batch size of 256 using standard cross-entropy loss. There is no early stopping, no learning-rate scheduler, and no explicit regularization beyond that symbolic dropout layer. The goal was speed and compliance, not virtue.

“I wanted a model that learns quickly, not wisely.”

What the Model Learns (And What It Doesn't)

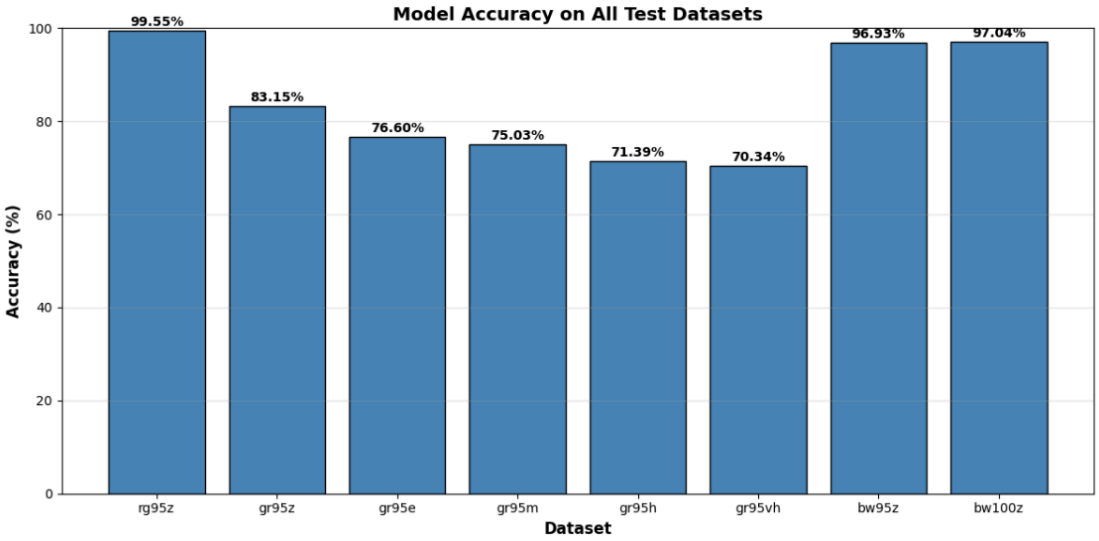
On datasets like **RG_95Z** and **GR_95Z**, the model performs exceptionally well near-perfect accuracy, rapid convergence, and confident predictions. At first glance, it appears successful. But this success is misleading.

Given a training distribution where color and label are aligned 95% of the time, the easiest solution is to exploit color statistics available directly in the input channels. The early convolutional filters latch onto channel-wise intensity patterns, and downstream layers amplify these correlations. Pooling layers further exacerbate this behavior. While they provide spatial invariance, they also aggressively discard fine-grained stroke information precisely the cues required for genuine digit recognition. What remains is a coarse, color-dominated representation that works beautifully until color is removed or perturbed.

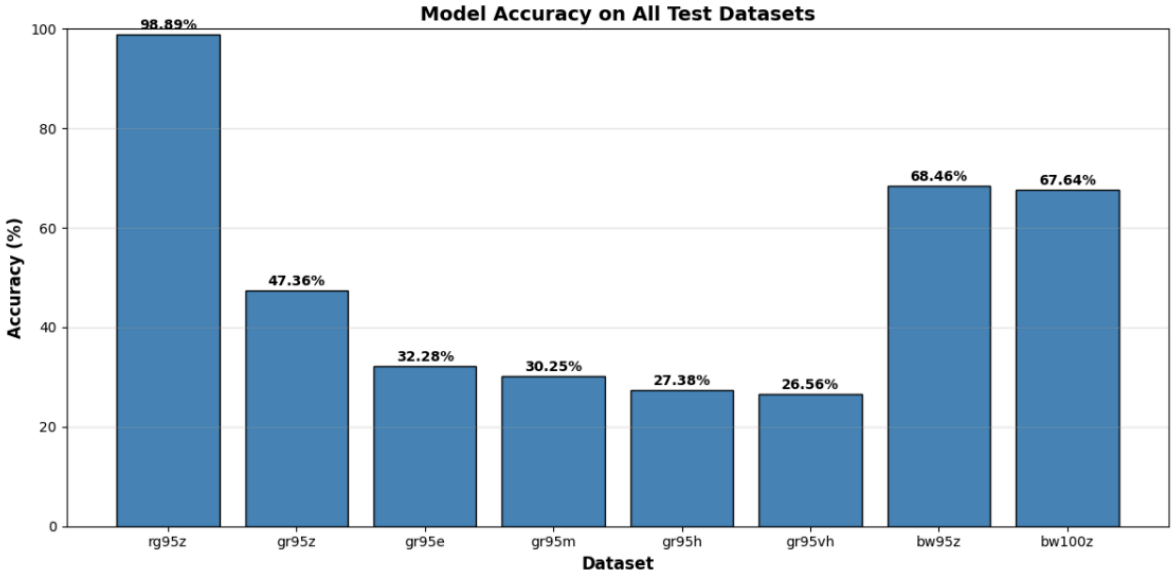
When evaluated on datasets where color bias is weakened or eliminated such as GR_95M, GR_95H, or especially BW_100Z performance collapses. Accuracy drops sharply, confidence becomes misplaced, and predictions deteriorate in exactly the scenarios where color is no longer informative.

I tried lots of approaches, changing the epoch, learning rate(Lr), batch size (Batch), dropout, etc to find which set of parameters leads to the worst accuracy.

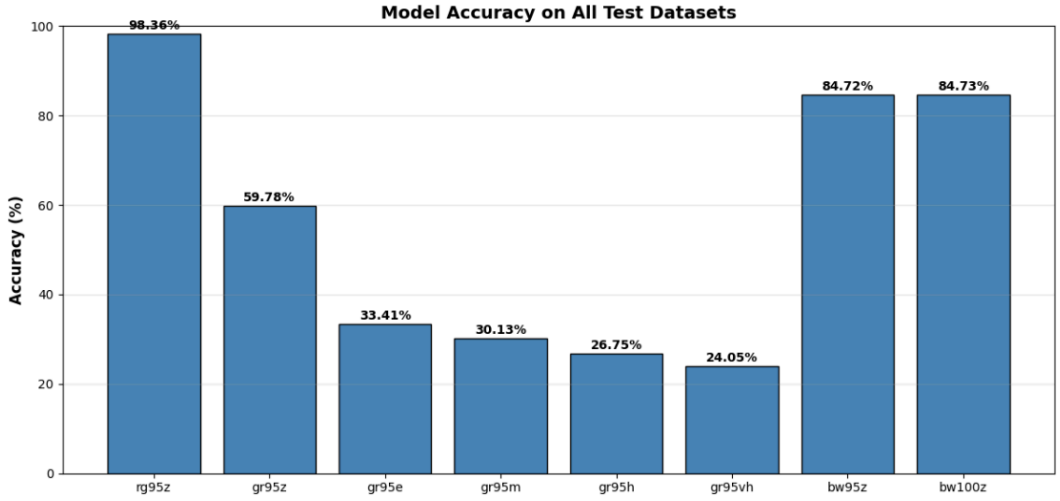
Lr = 0.001



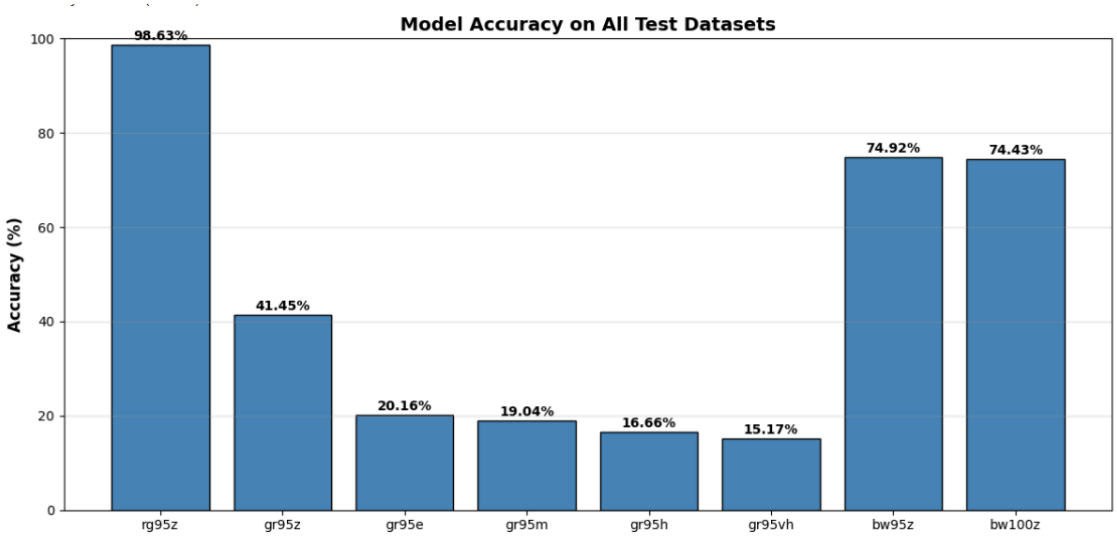
Lr = 0.01



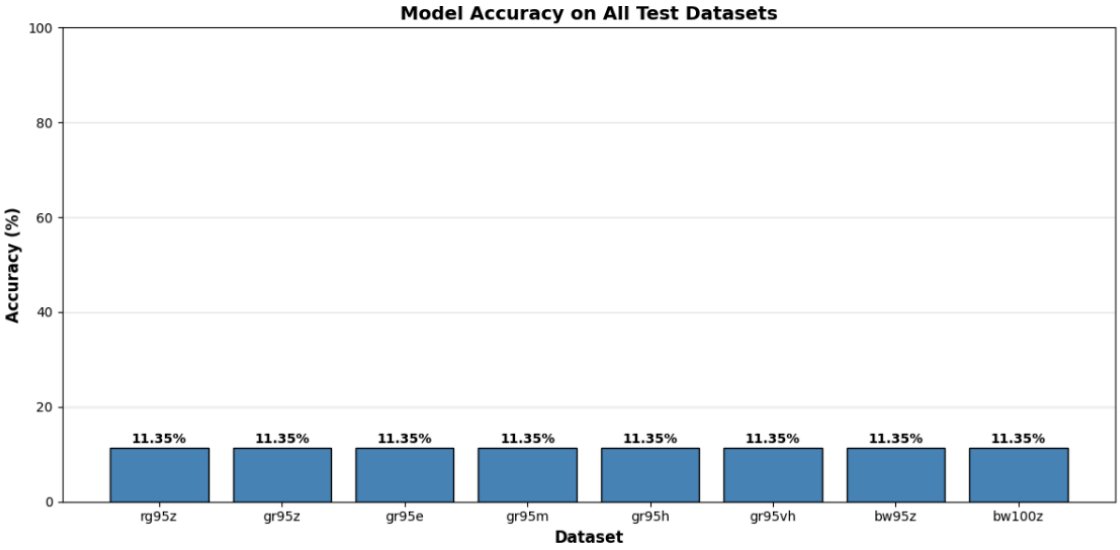
Batch = 64lr = 0.01



Lr = 0.02


Model Accuracy on All Test Datasets
Accuracy (%)
100
80
60
40
20
0
98.63%
41.45%
20.16%
19.04%
16.66%
15.17%
74.92%
74.43%
rg95z
gr95z
gr95e
gr95m
gr95h
gr95vh
bw95z
bw100z

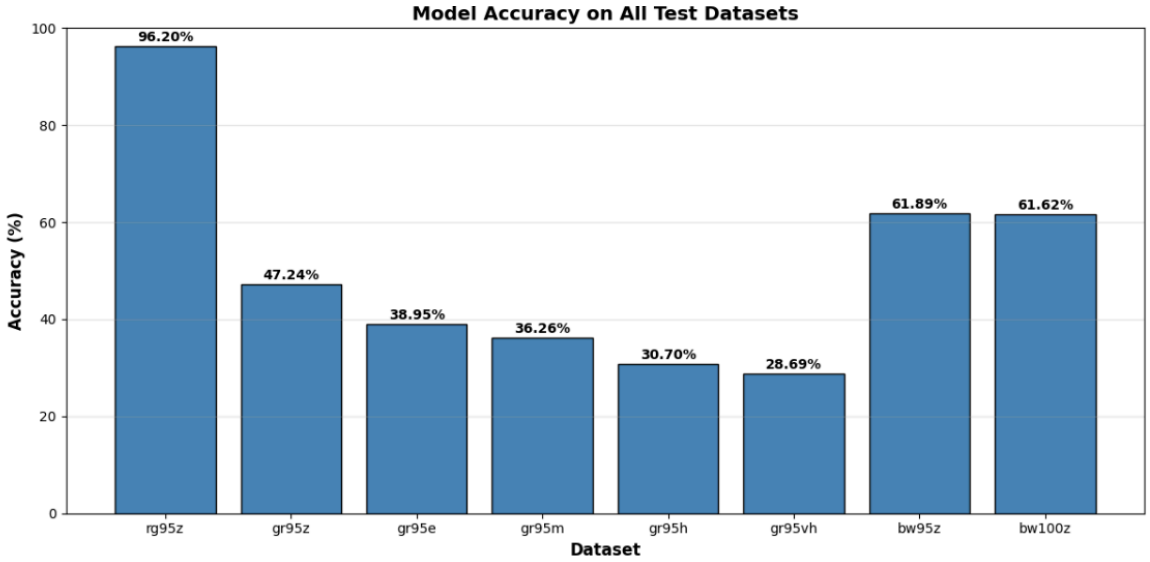
Lr = 0.03


Model Accuracy on All Test Datasets
Accuracy (%)
100
80
60
40
20
0
11.35%
11.35%
11.35%
11.35%
11.35%
11.35%
11.35%
11.35%
rg95z
gr95z
gr95e
gr95m
gr95h
gr95vh
bw95z
bw100z
Dataset

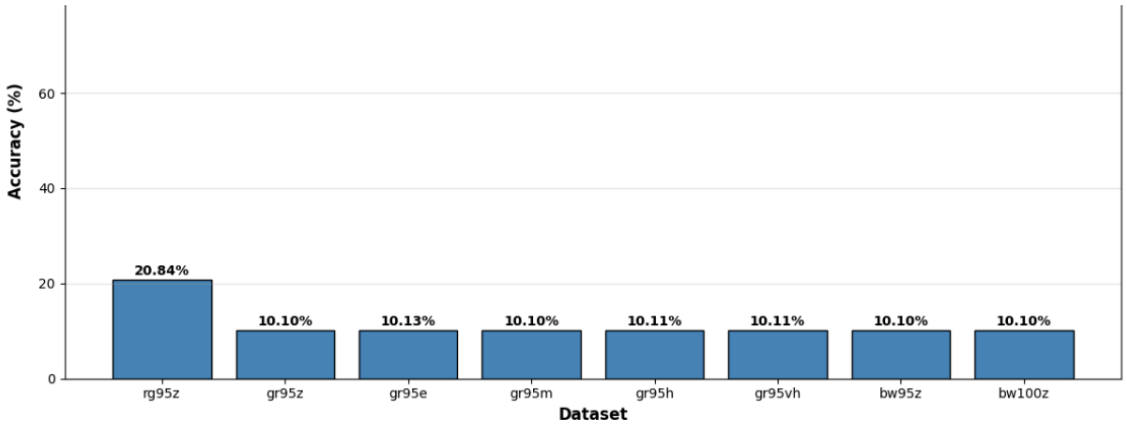
Try try but dont cry

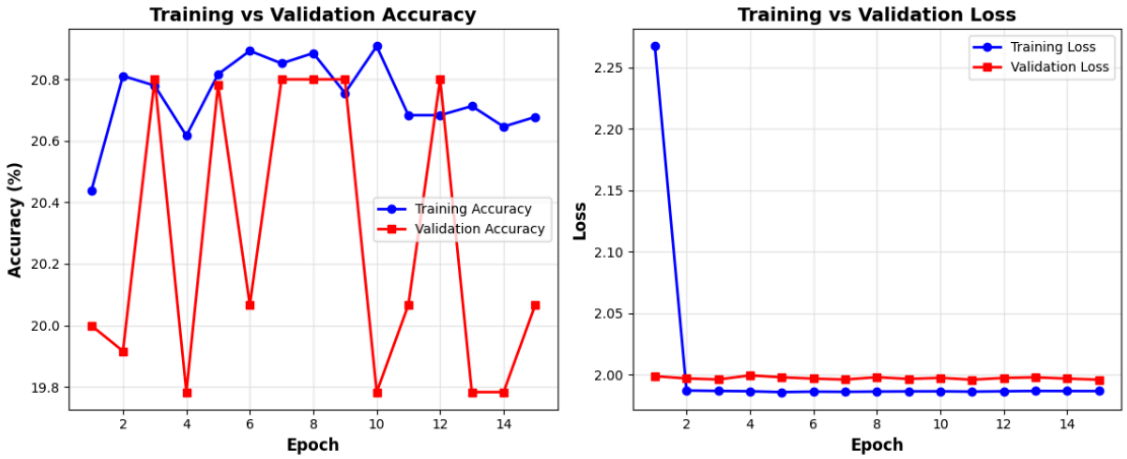
Despite the set back in the previous iteration I continue testing.

Lr = 0.03 dropout = 0.0



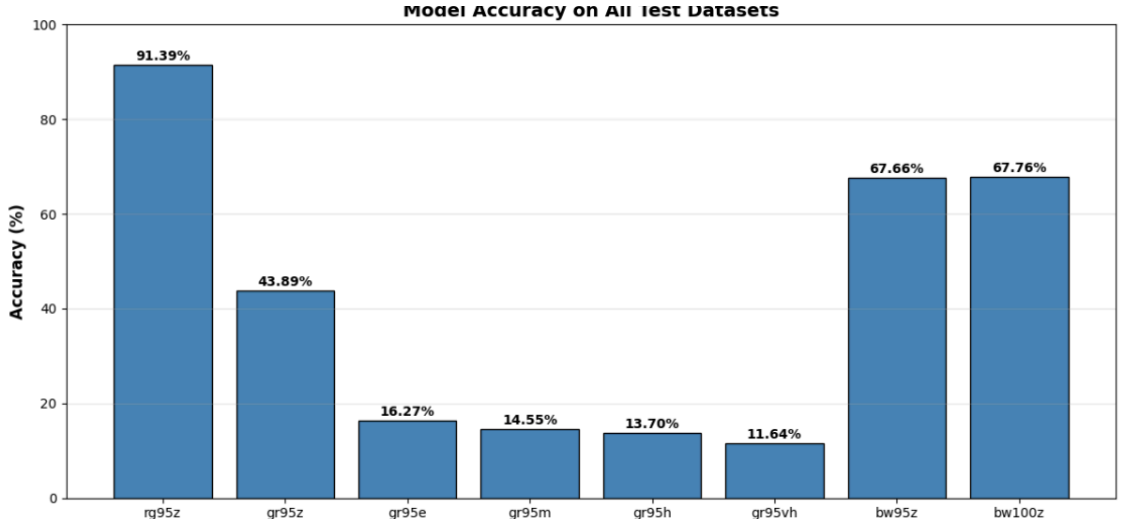
Lr 0.045 dropout 0.0 epoch 15The



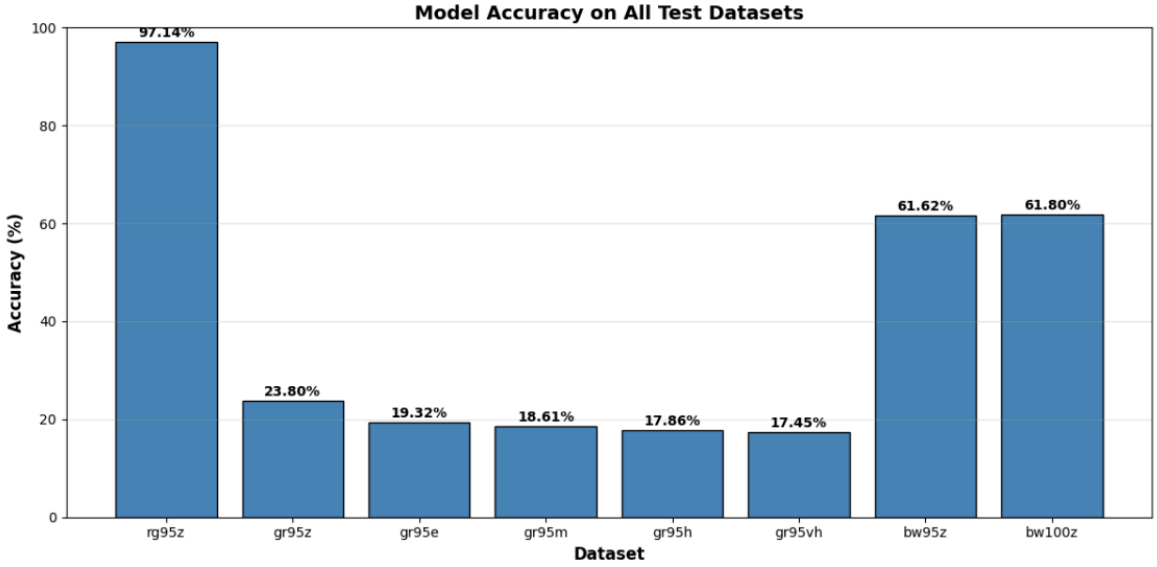


The training vs validation accuracy shows that there is no convergence, thus the poor accuracy.

Lr 0.045 epoch 15 dropout 0.05 batch 128

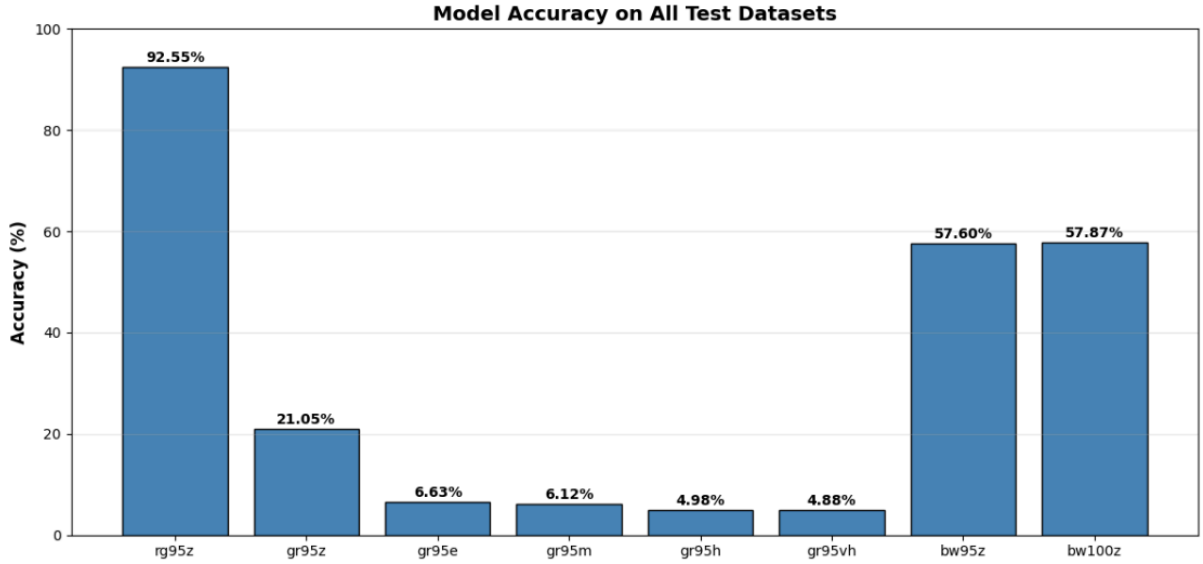


Lr 0.045 epoch 15 dropout 0.05 batch 256



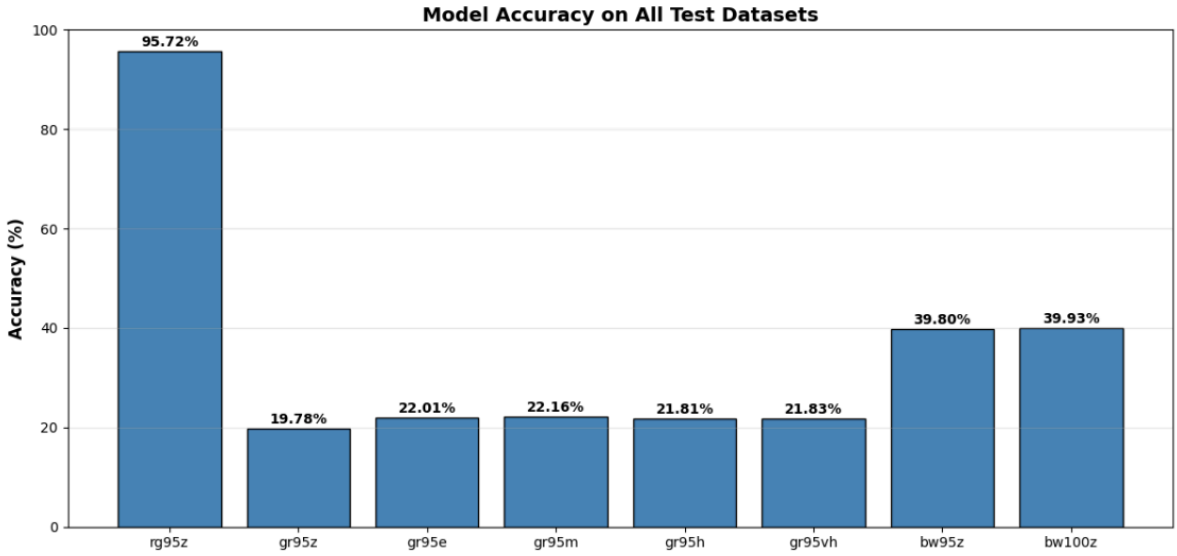
Voila !

Same as above but 50 epoch



The model has failed miserably for reversed biases.

Same as above but dropout 0.1 and epoch 35



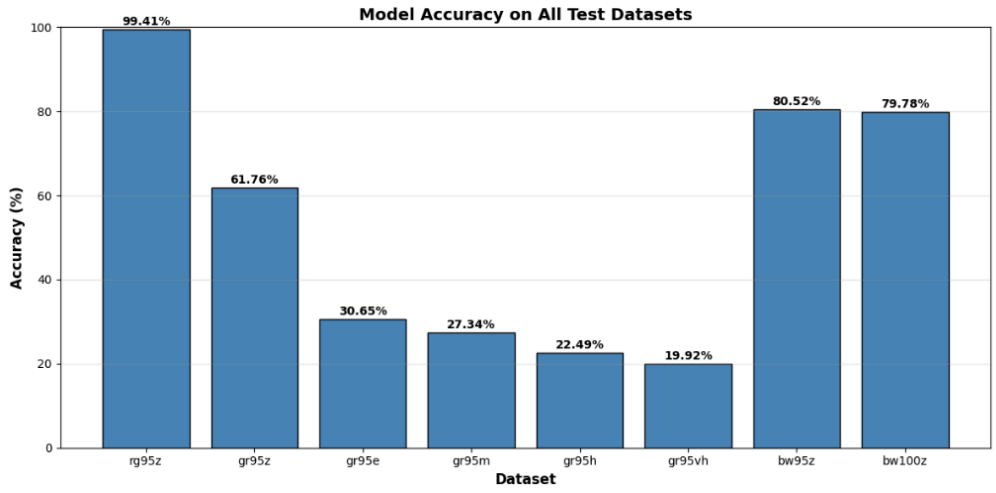
Smudging the Paint to Find the Drawing

The first attempt at resisting shortcut learning was intentionally minimal. Rather than redesigning the architecture or introducing explicit regularization, I chose to bias the model at the point where shortcuts usually form first. Early convolutional layers tend to lock onto the easiest available signal, and in the case of Colored MNIST that signal is overwhelmingly chromatic. The goal here was not to remove color, but to make it less immediately exploitable so that structural cues had a chance to surface.

Hybrid Gaussian Filters

The first modification explored was the use of Gaussian filters as a fixed preprocessing operation embedded directly within the network. Convolutional neural networks typically learn their earliest filters freely, but this flexibility often leads to the emergence of color detectors instead of shape or edge detectors when color correlates strongly with labels. To counter this, the first convolutional layer was initialized and fixed with a Gaussian kernel, effectively converting it into a low pass filter.

The underlying hypothesis was that Gaussian blurring suppresses high frequency details while preserving coarse spatial structure. By smoothing the input before any learned feature extraction occurs, the network is nudged toward broader shapes and spatial layouts rather than sharp pixel level variations or color specific edges. This architectural bias aims to reduce sensitivity to noise and color artifacts while encouraging the model to rely more heavily on shape based cues. The weights of the first convolutional layer were manually overwritten with this normalized Gaussian kernel and kept fixed throughout training. During the forward pass, the input image is first convolved with these Gaussian weights, applying a controlled blur within the model itself rather than as an external preprocessing step.



Lets Play 3D Polling

The second modification addressed how color information is handled during spatial downsampling. Standard pooling layers reduce spatial resolution independently per channel, but learned features across channels often compete implicitly. When one channel encodes a strong shortcut signal, its activations can dominate and suppress information in other channels, reinforcing color dependence.

To mitigate this, a channel wise pooling strategy was introduced. The hypothesis here was that by pooling each color channel independently, the model can preserve channel specific activations during downsampling without allowing one color to overshadow the others. This encourages a more balanced representation where structural information present across channels is less likely to be drowned out by a single dominant chromatic feature.

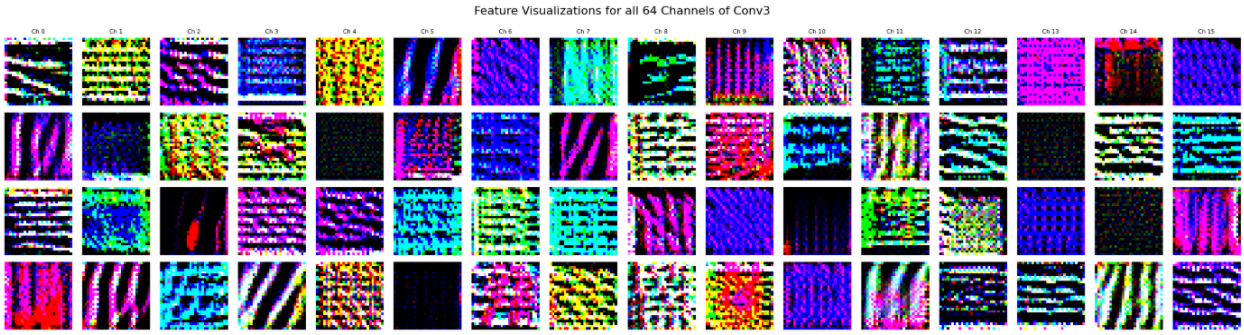
A custom ChannelWiseMaxPool2d layer was implemented to replace all standard max pooling layers in the three layer CNN. The input tensor is processed channel by channel, with each channel undergoing a standard two dimensional max pooling operation across spatial dimensions. The pooled channels are then concatenated back together, preserving the original channel count while reducing spatial resolution.

Together, these two interventions introduce controlled constraints rather than aggressive corrections. The Gaussian filter limits what the model initially sees, while channel wise pooling regulates how information competes as it flows deeper into the network. Neither guarantees robustness on its own, but both serve to slow the model's instinctive reliance on color, giving shape and structure a better chance to influence learning.

The Prober

My initial approach (https://www.kaggle.com/code/akshatatkaggle/task2-app1) was to apply a predefined interpretability method, then probe the model directly and iteratively, identifying failure modes as they appeared and modifying the analysis accordingly. As the framework evolved, I realized that many of the solutions I arrived at independently converge to principles that have since been formalized in the interpretability literature.

To reveal what the network is actually "thinking," I defined a CNN3Layer architecture featuring three convolutional blocks, each with max-pooling and ReLU activation, followed by two fully connected layers. I then transitioned from training to investigation by loading pre-trained weights from a .pth file, freezing them to prevent changes. The core of this visual exploration is the visualize_neuron function, which employs a gradient ascent optimization technique to effectively "reverse-engineer" the model's preferences. Starting with a blank, zero-filled image, the function uses the Adam optimizer to iteratively adjust the input pixels until the mean activation of a specific channel in a chosen convolutional layer is maximized. This process generates a very very abstract grid of pixels which is supposed to activate the particular neuron the most.



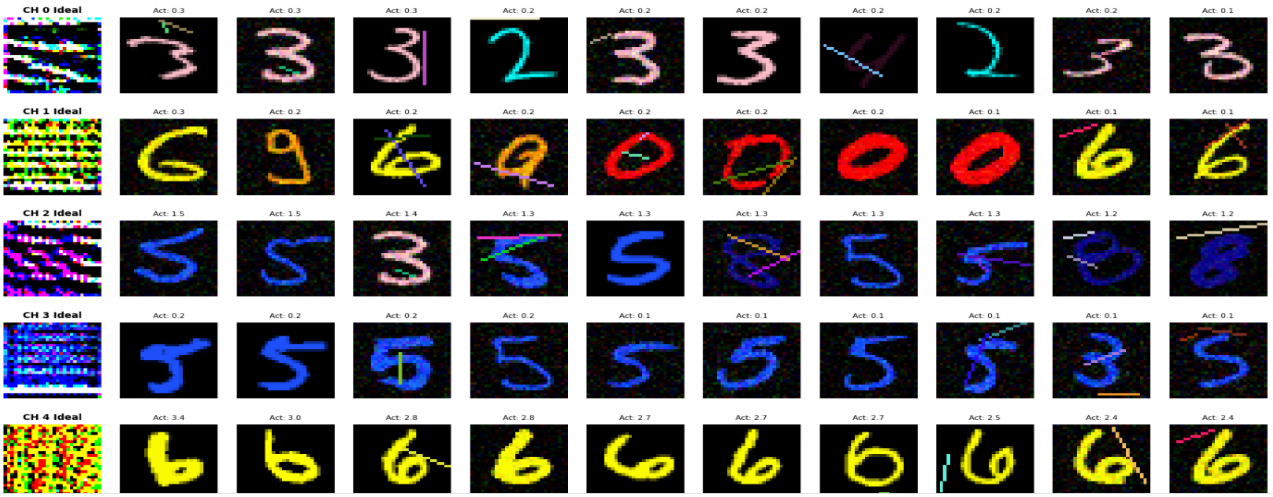
Honestly I was surprised to see something like this. I was like, what the hell does this mean?

Imagine if some alien organism landed on Earth and could do these things.

Everybody would be falling over themselves to figure out how… And so really the thing that is calling out in all this work for us to go and answer is, “What in the wide world is going on inside these systems??” by Chris Olah

This is the output I got in my initial approach, say you have pink longitudnal lines, then that neuron has learnt the feature of ‘standing pink line’ which can occur in a pink 1 or pink 7.

To visualise better I generated top 10 images that activate the channels in layer 3 the most.



Notice how the first channel (ch 1) is activated by yellow 6, orange 9, red 0. It is detecting curves of shades of Red. Where as the (ch 3) detects mostly only blue 5. Why ? Because ch1 is in a super position between yellow 6, orange 9 and red 0. That means it is a polysemantic neuron.

To learn more about this I started search for sources online and this distill article really stood out with me https://distill.pub/2017/feature-visualization/.

Approach 2 (Improved approach 1)
(https://www.kaggle.com/code/akshatatkaggle/task2-approach2)

Noise is not all bad

My initial attempts to visualize_neuron in the notebook were a bit of a disaster, lots of high-frequency noise and brittle pixel-level artifacts. I realized the gradient ascent was too sensitive, so I progressively introduced spatial jitter (for translational stability) and total variation regularization (based on the distill article) to force the model to find translationally robust patterns instead of just overfitting to noise. Disabling dropout during the process was another thing that helped the Adam optimizer.

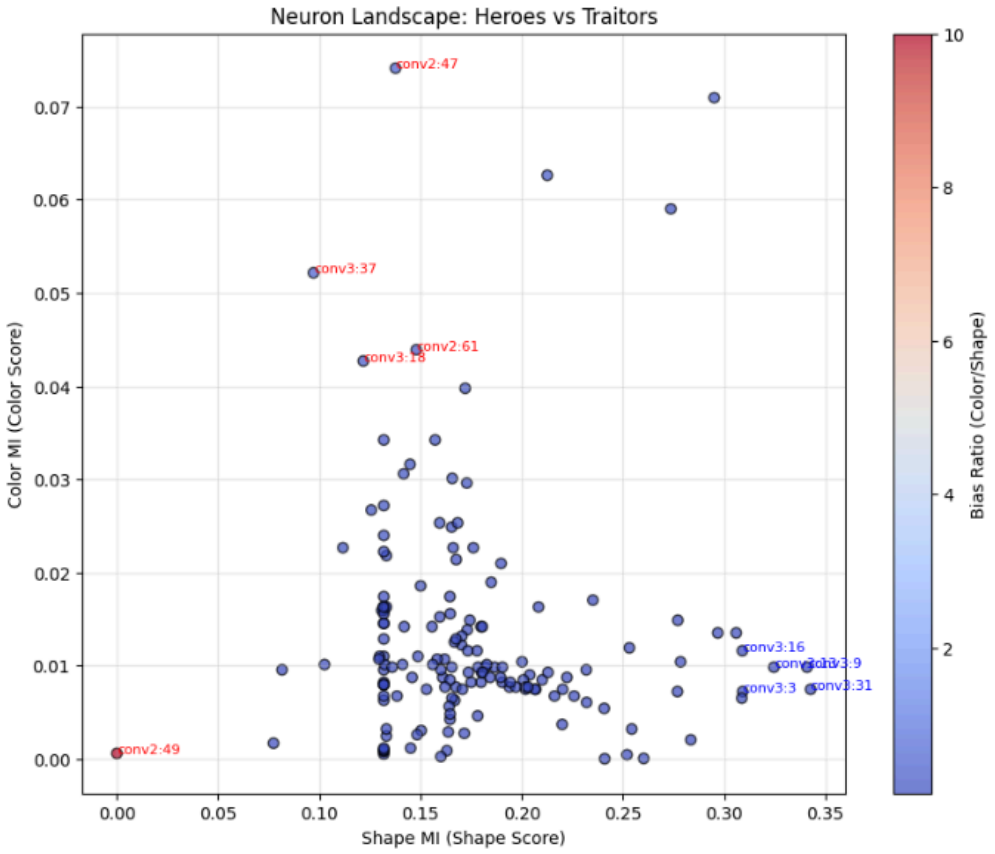
Shape of you

Since color is so heavily correlated with labels in the biased set, any neuron triggered by red is going to look "confused" across multiple digit classes by default. To get a real measure of feature entanglement, I shifted my focus to shape polysemanticity using the grayscale dataset. This ensures that any multi class activation actually reflects an ambiguity in the digit's structure, like a curve shared between a "3" and an "8", rather than just a shortcut colored pixel.

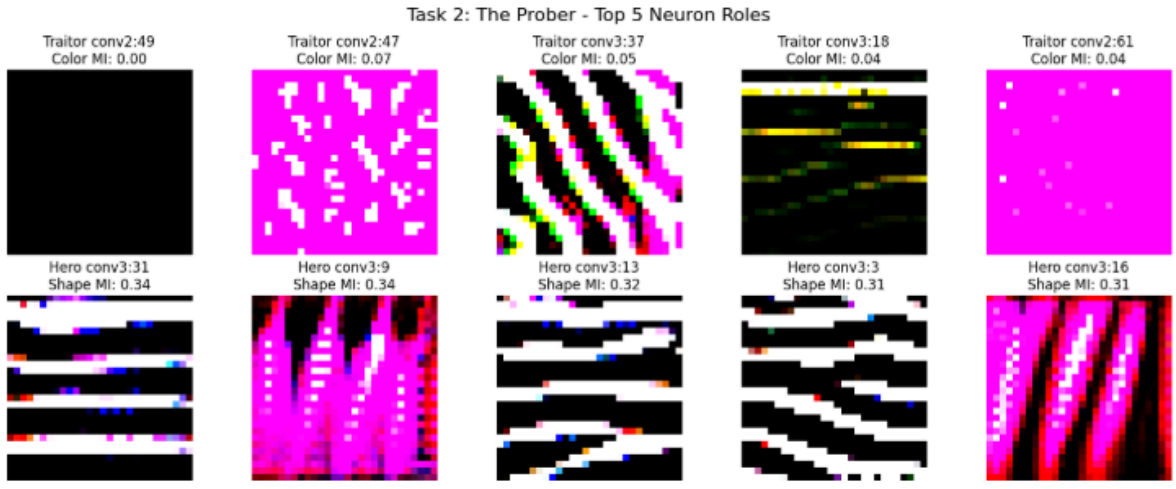
Finding Brutus

I measured mutual information with digit identity. Initially, I binarized activations using fixed thresholds, but I noticed that the resulting mutual information estimates were unstable and overly sensitive to noise, particularly for neurons with low activation variance. I suspected this could become a serious issue, so I switched to quantile-based discretization of activations, which produced far more consistent neuron rankings. Only later did I realize that this dataset-level attribution of neuron concept relationships converges closely to approaches such as Network Dissection, where neurons are evaluated using global statistics rather than isolated examples.

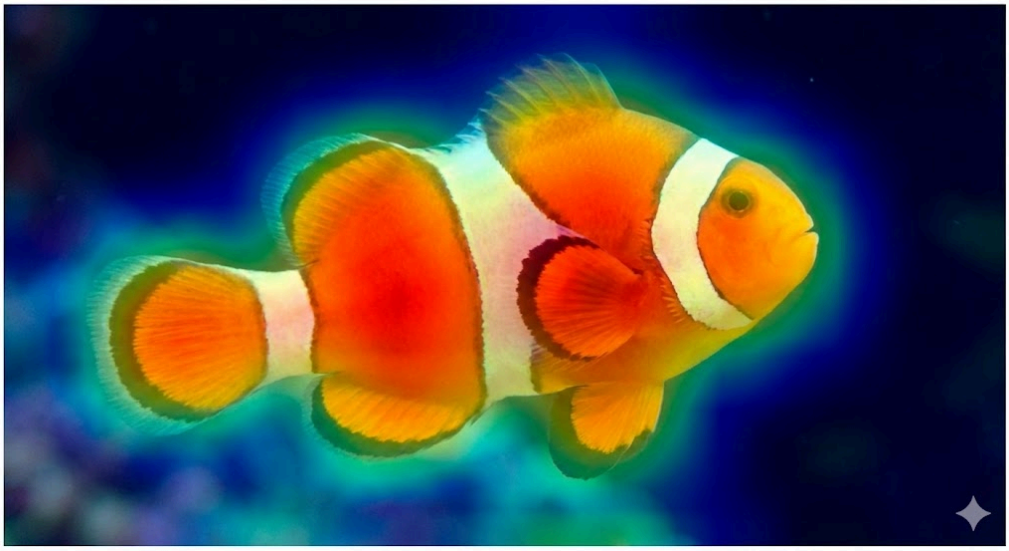
This mutual information analysis revealed a clear separation between neurons that strongly encoded color and those that encoded digit shape. Some neurons exhibited high MI with color on the biased dataset while collapsing on grayscale data, whereas others retained high MI with digit identity even when color information was removed. I labeled these two populations as “Traitors” and “Heroes,” respectively. The term Traitor refers to neurons that exploit spurious color label correlations to achieve apparent performance gains, while Heroes refer to neurons that encode invariant, task-relevant structure. These labels are not anthropomorphic but serve as concise descriptors of two qualitatively different representational roles that naturally emerge in shortcut-prone training regimes.



Top 5 Traitors and Top 5 Heroes

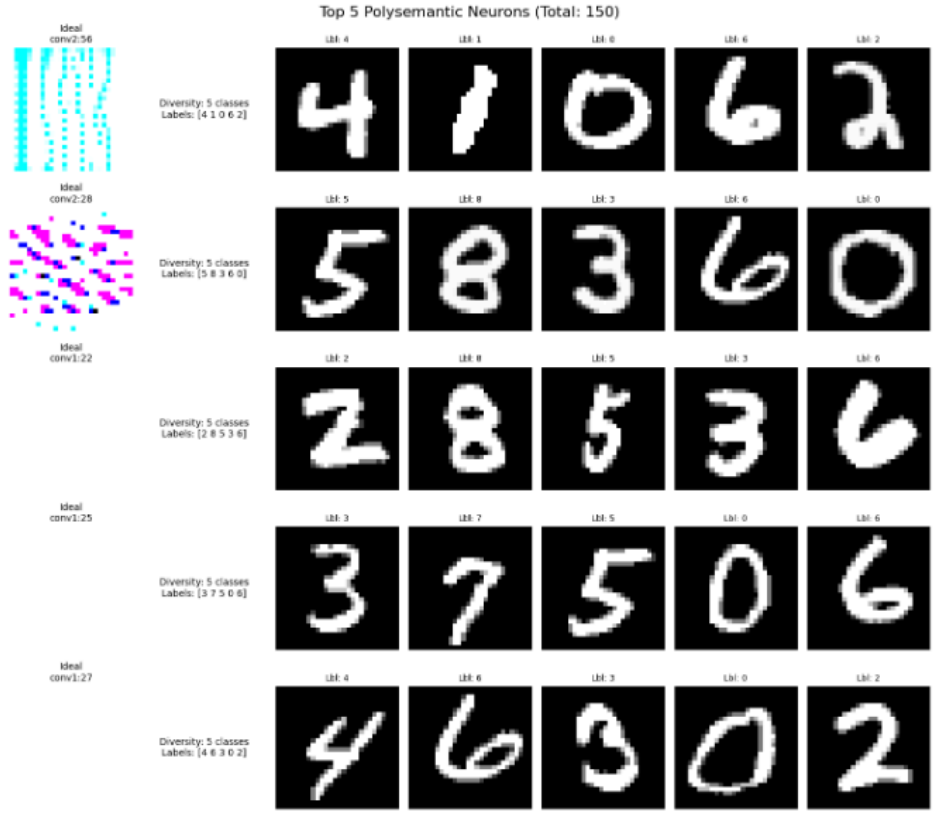


Finding ~~Nemo~~ Polysemantic Neurons

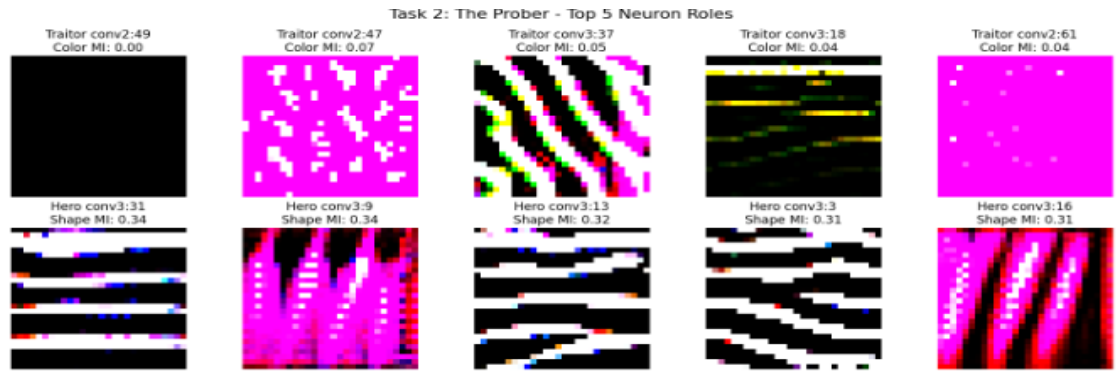


PS - Nemo but with gradcam

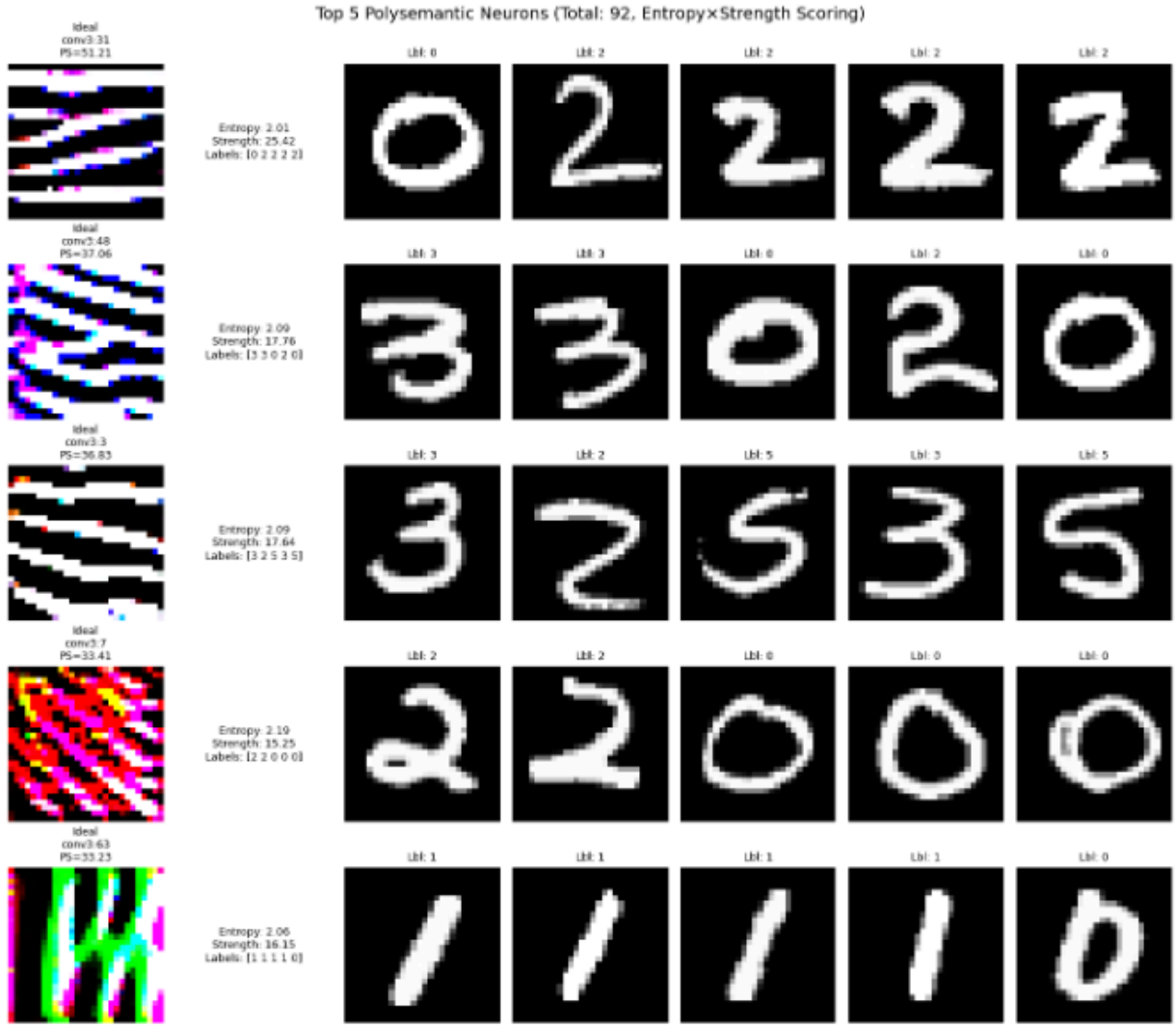
Several neurons appeared to activate for many different digit classes, which initially led me to interpret them as polysemantic. However, upon closer inspection, many of these neurons produced nearly blank or incoherent visualizations and exhibited uniformly weak activation across the dataset. I realized that neurons that barely activate at all will naturally show multiple labels among their top-activating samples, even if they encode no meaningful feature. This led me to suspect that naive top-k label counting conflates semantic ambiguity with inactivity, creating an “inactive neuron illusion.”



To resolve this, I reasoned that semantic diversity should only be measured when a neuron is genuinely active. I introduced an activation strength criterion based on robust statistics such as the mean of top k spatial activations summarized by a high percentile across the dataset. Neurons with negligible activation strength were excluded entirely. For neurons that did activate meaningfully, I computed the entropy of digit labels only among inputs that produced strong activation. This led me to define a joint polysemanticity metric, which I refer to as the “Polysemantic Score,” defined as the product of label entropy and activation strength. This formulation naturally penalized inactive neurons while highlighting neurons that activated strongly across multiple digit classes. After formalizing this score, I later realized that the same principle, conditioning semantic diversity on activation magnitude, is central to recent work on polysemantic neuron analysis and feature disentanglement (2).



Now I seemed to have got better Polysemantic neurons



The Interrogation

For this next phase of the investigation, I began by referencing the foundational Grad-CAM paper, "Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization," (3) which provides the technical framework for understanding a model's "focus". While exploring how to apply this to my specific digit classification problem, I discovered a Medium article on implementing Grad-CAM in PyTorch and decided to follow a similar approach for my own implementation(4). I felt that simply knowing the model's accuracy wasn't enough; I needed to physically see if the model was making decisions based on the actual digit or just taking a shortcut through the background color.

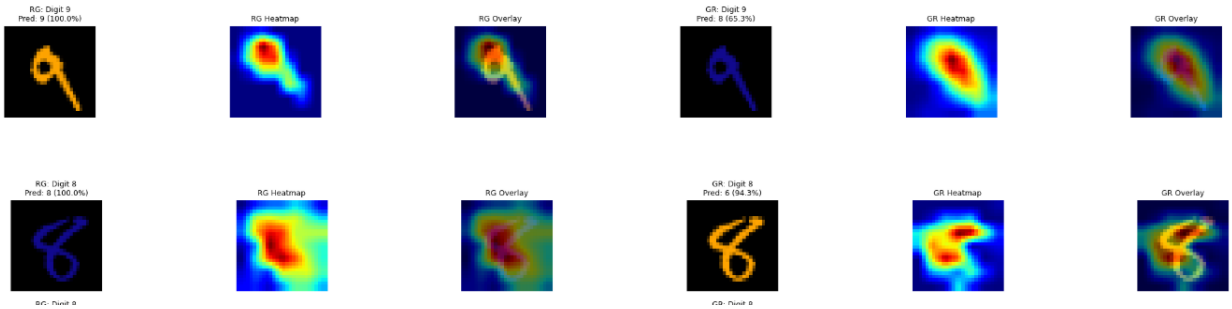
The methodology I implemented in the notebook involves capturing the gradients of a specific target class with respect to the feature maps of the final convolutional layer. I set up specific hooks to intercept the activations during the forward pass and the gradients during the backward pass. By globally average-pooling these gradients, I calculated a weight for each feature map and combined them using a ReLU activation to generate a coarse heatmap of the most important regions. I then upsampled this heatmap to match the original input image size so I could overlay it and see exactly which pixels were driving the model's prediction.

The expected result was that a biased model would show high-intensity heatmaps focused almost entirely on the background color rather than the number itself. When I visualized the actual results, my intuition was confirmed; for the biased model, the Grad-CAM maps were often diffused across the colorful background, showing that the "Traitors" in the network were ignoring the digit contours. However, for the models that were forced to learn shape, the heatmaps were concentrated precisely on the strokes of the digits.

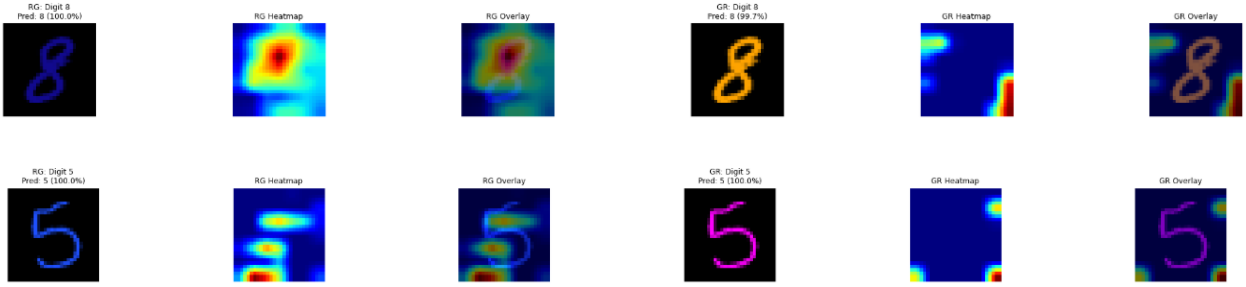
The conclusion from these visualizations is that Grad-CAM is an essential tool for unmasking the "shortcut learning" behavior in biased networks. It allowed me to see that even when the model was getting the answer "right," it was often doing so for the "wrong" reasons by looking at the color instead of the shape. This proof of focus confirmed that the model's reliance on color "Traitors" was a physical reality in the convolutional layers, providing a clear path for where I needed to perform "digital surgery" to make the model more robust.

Model looking at wrong place from start -

Despite looking at right place, because of color wrong prediction -



Correct prediction in reversed dataset, but looking at wrong place -



The above visualisations help us understand that the true accuracy in reversed dataset from task1 is much lower than what it is measured, further proving our claim of model learning shortcuts instead of shape.

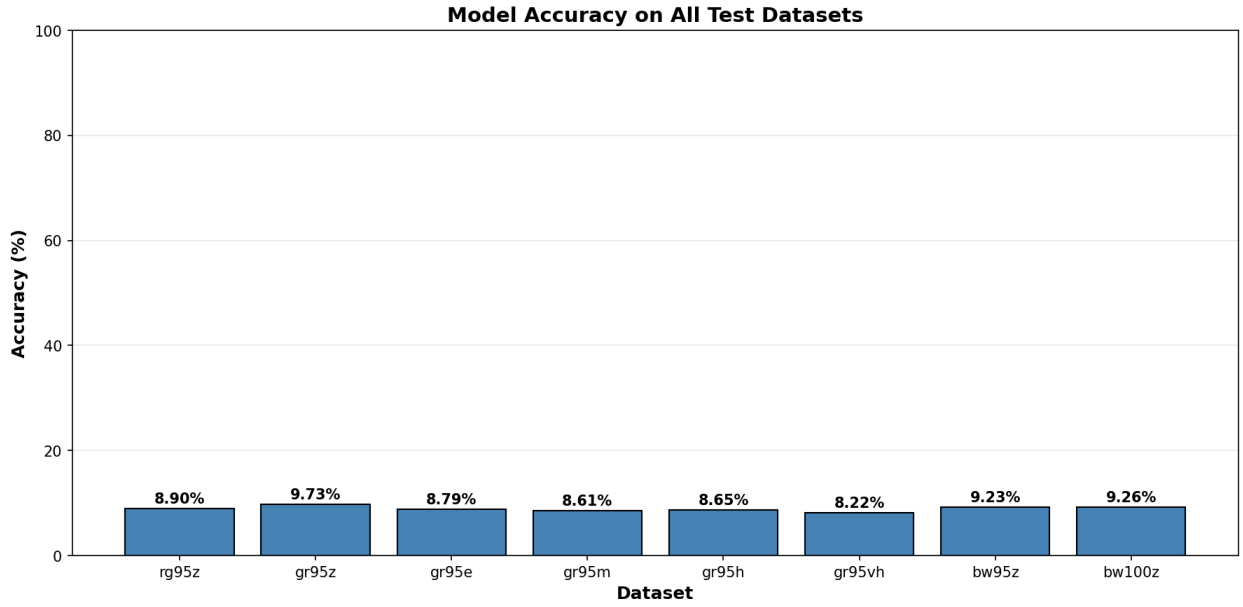
Link to the notebook - https://www.kaggle.com/code/akshatatkaggle/task3-app1

The Intervention

Expect the Unexpected

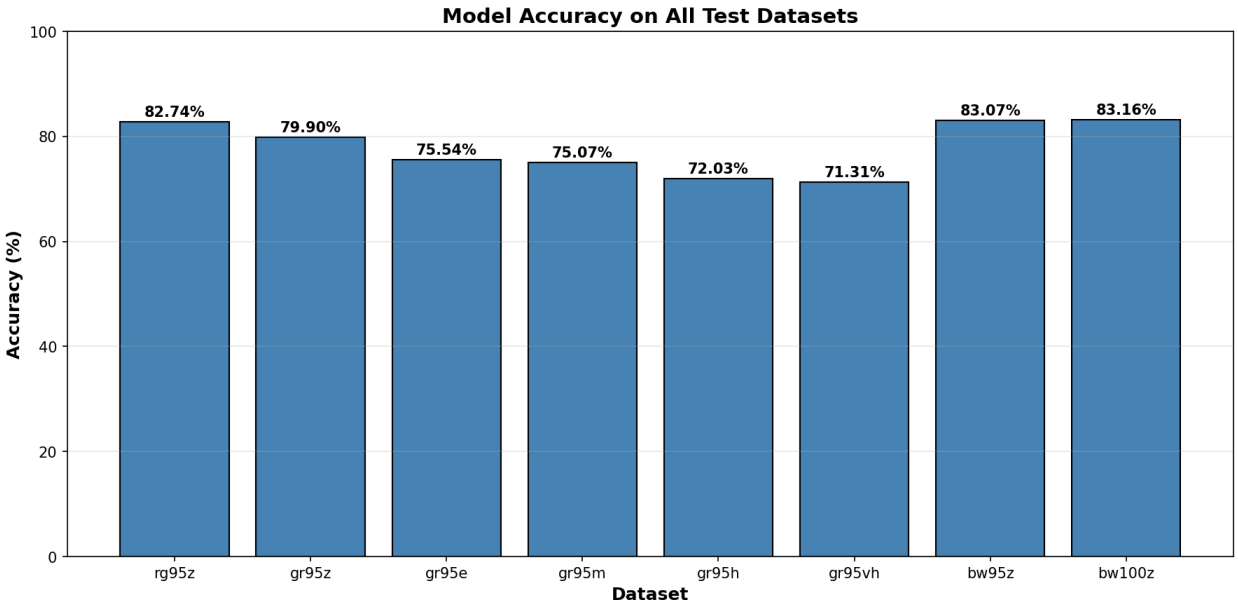
After going through various approaches suggested online, most of them were about changing the dataset or manipulating the dataset, but this couldnt be done due restraints of this task. I settled to use IRM (Invariant risk management) after reading this article https://medium.com/data-science/how-to-make-deep-learning-models-to-generalize-better-3341a2c5400c.

From my previous experience with hyperparameter, I wanted to know how to properly balance the classification risk and the invariance penalty. I initially started with a penalty of 10000, which led to very bad performance as the model seemed unable to learn anything at all.

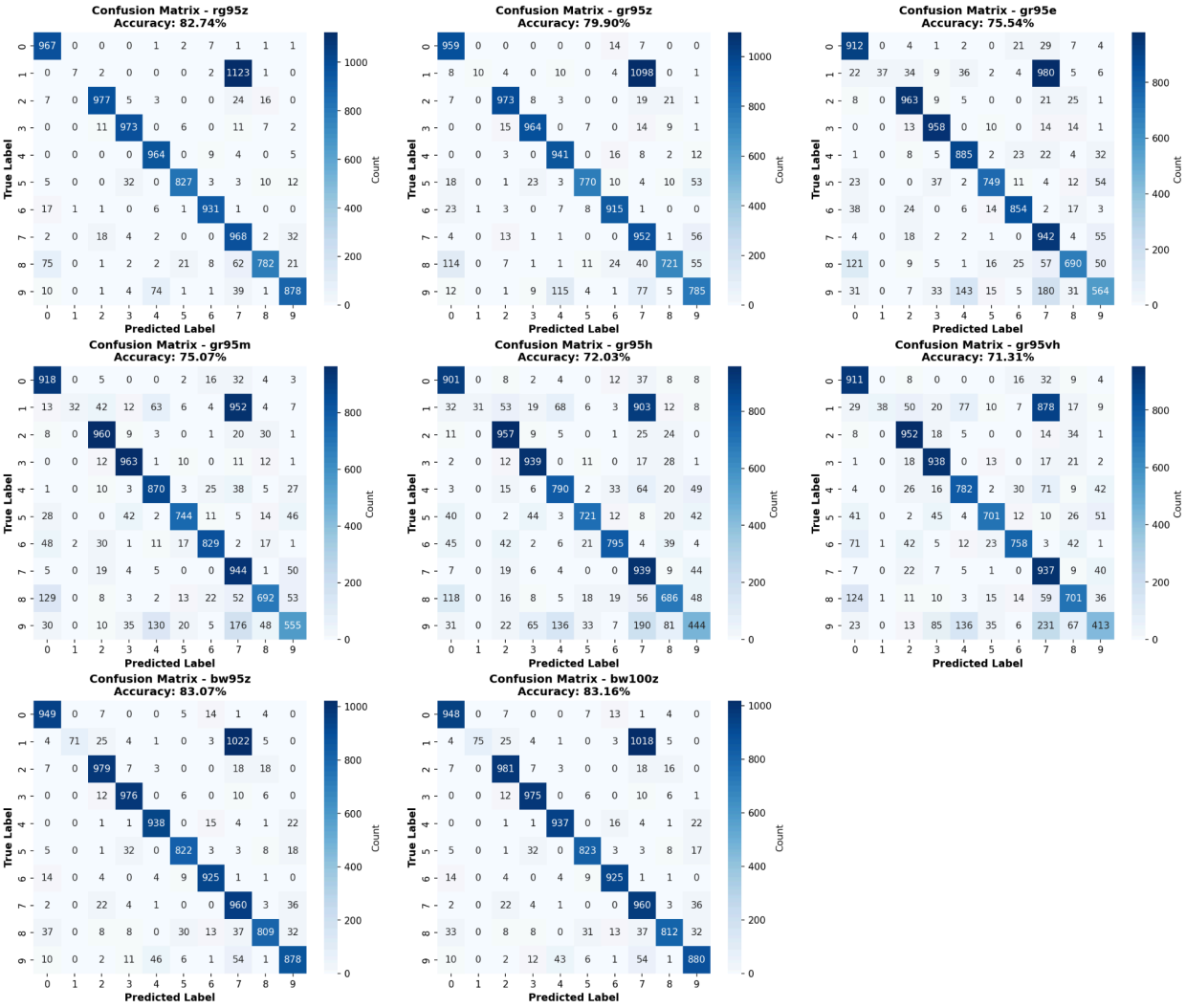


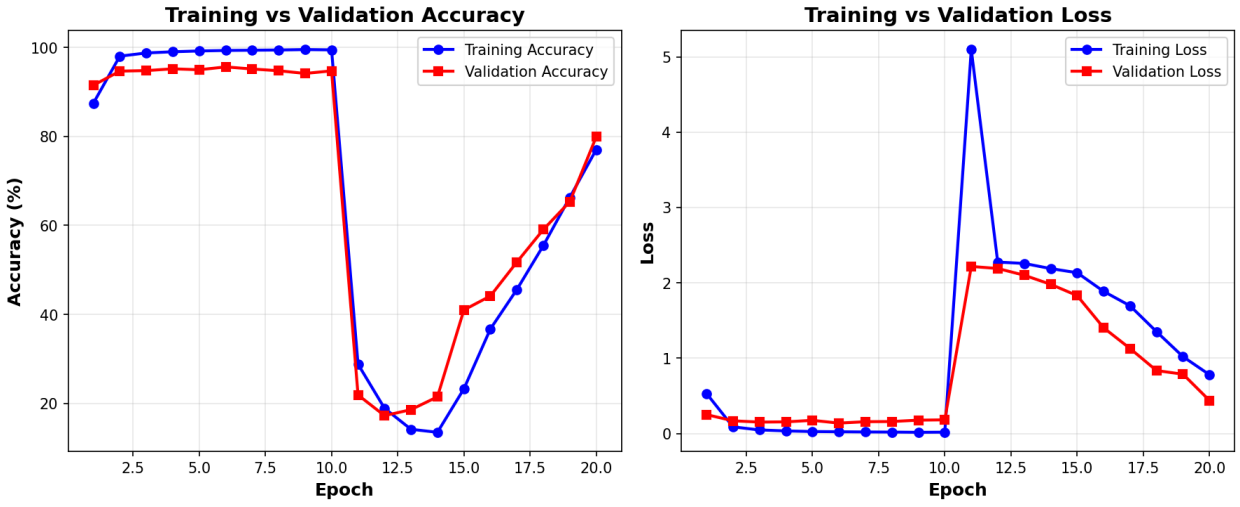
Divine intervention

As I experimented with hyperparameters further, I thought that the penalty was too harsh and reducing it to 100 was a turning point, as I finally saw a drastic improvement in performance.

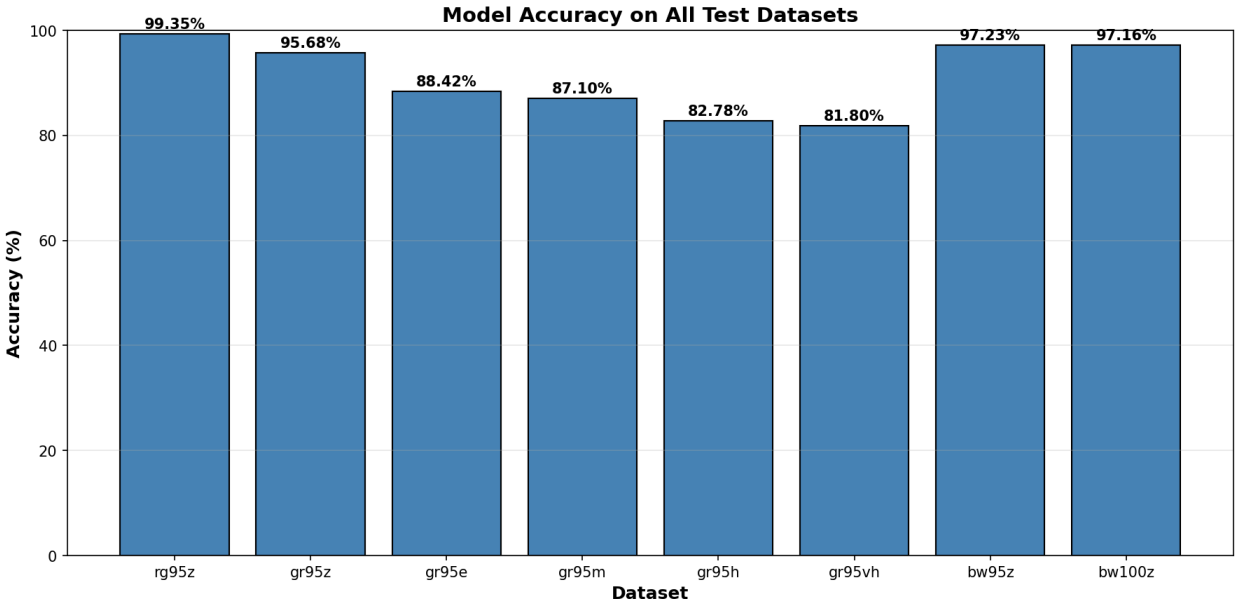


Despite its sensitivity to hyperparameters, IRM turned out to be very useful for defining what "robustness" actually means in this project. The early models were almost entirely dependent on color, but as I refined the IRM training, the model started to maintain some level of accuracy even when the colors were flipped in the hard test set. This contrast was one of the first indicators that the model was moving beyond superficial shortcuts and starting to focus on the more difficult, invariant geometry of the digits.


Confusion Matrix - rg95z
Accuracy: 82.74%
Confusion Matrix - gr95z
Accuracy: 79.90%
Confusion Matrix - gr95e
Accuracy: 75.54%
Confusion Matrix - gr95m
Accuracy: 75.07%
Confusion Matrix - gr95h
Accuracy: 72.03%
Confusion Matrix - gr95vh
Accuracy: 71.31%
Confusion Matrix - bw95z
Accuracy: 83.07%
Confusion Matrix - bw100z
Accuracy: 83.16%
True Label
Predicted Label
Count


Training vs Validation Accuracy
Training Accuracy
Validation Accuracy
Accuracy (%)
Epoch
Training vs Validation Loss
Training Loss
Validation Loss
Loss
Epoch

The sudden spike and dip is the epoch from when annealing ends and irm starts. This made me think if increasing the number of epochs would led to better results. Nevertheless I decided to try it



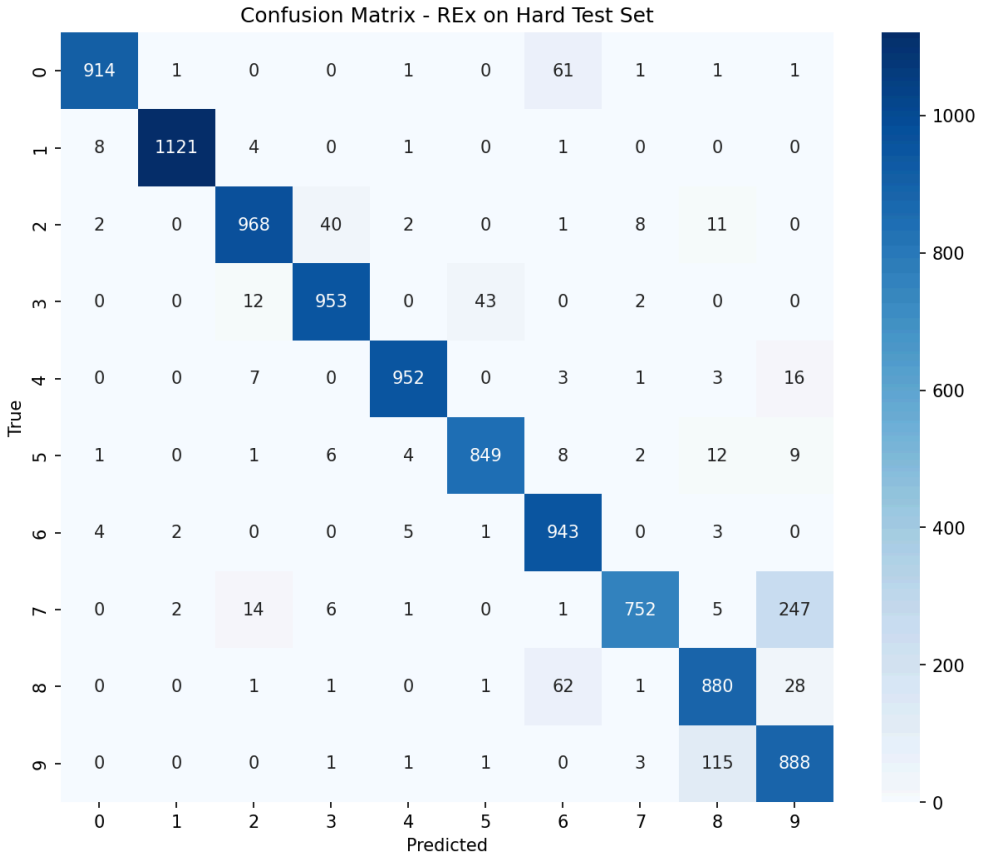
Interestingly Increasing number of epoch led to increasing the accuracy of the model.

Second Opinion is always recommended

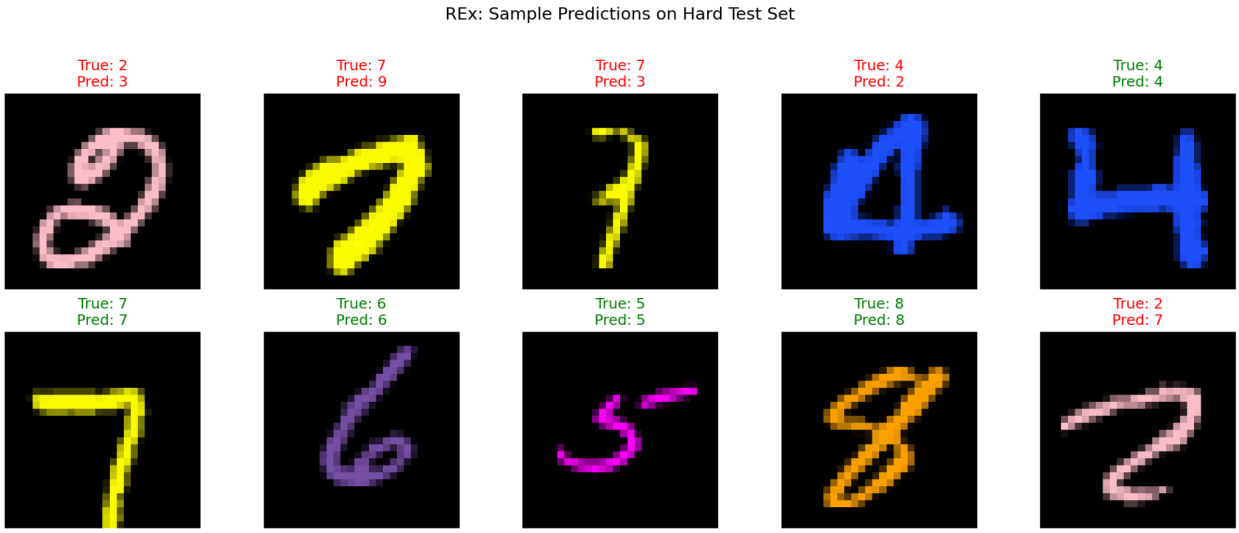
I didn't have any solid reason to choose REx (Risk Extrapolation) (5) at the beginning. I was reading through different robustness methods, mostly out of curiosity, and came across its methodology. What caught my attention was how simple the idea felt. Instead of forcing invariance explicitly, it just asked the model to avoid behaving very differently across environments. Intuitively, that made sense to me. If a feature only works in one setting, the loss should fluctuate, and penalizing that fluctuation felt like a natural way to discourage shortcuts.

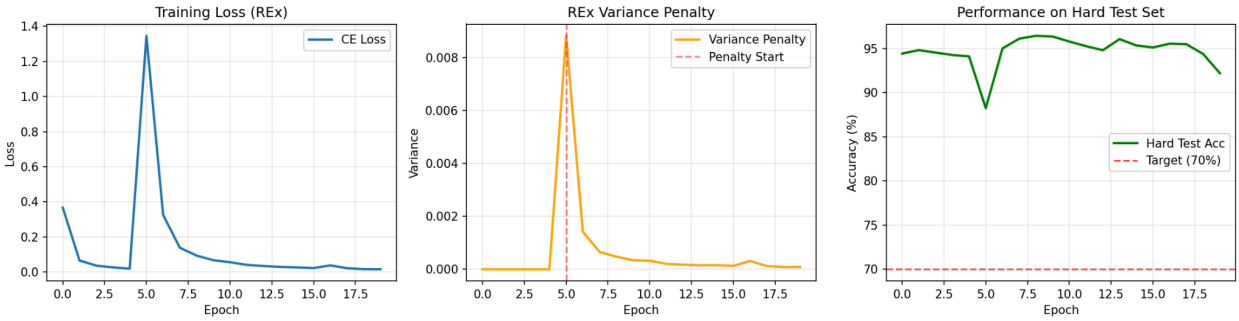
The algorithm itself is straightforward. The model is first trained normally using cross-entropy loss. After a few epochs, a variance penalty is added that penalizes differences in loss across environments. The final objective becomes a combination of average loss and the variance of that loss. Rather than enforcing hard constraints, REx softly nudges the model toward stable solutions.

To understand what was happening, I visualized the training loss, the variance penalty over epochs, hard test accuracy, a confusion matrix, and sample predictions. When the penalty was activated, the variance spiked briefly and then steadily collapsed.



Accuracy on the hard test set stabilized around 94 to 96 percent. The remaining errors were mostly between visually similar digits, which made the results feel grounded rather than accidental.





Again the spike indicates where the annealing stops and REx is implemented.

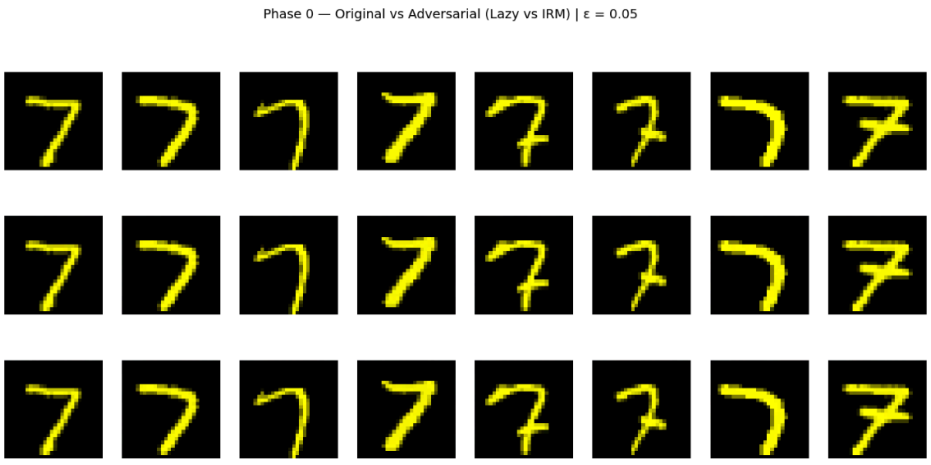
The Invisible Cloak

In this task my intuition was to take a noise pattern and modify it such that the model classifies the image incorrectly. After searching google, I came to know that FGSM (The Fast Gradient Sign Method) is a method that works on a very similar concept.. At first glance it feels almost too simple, and I remember being unsure whether such a crude, single-step attack would tell me anything meaningful. FGSM is one of the earliest adversarial attacks, introduced by Goodfellow et al. in 2015,(6) and it perturbs the input in the direction of the sign of the loss gradient scaled by a fixed epsilon.

Accuracy

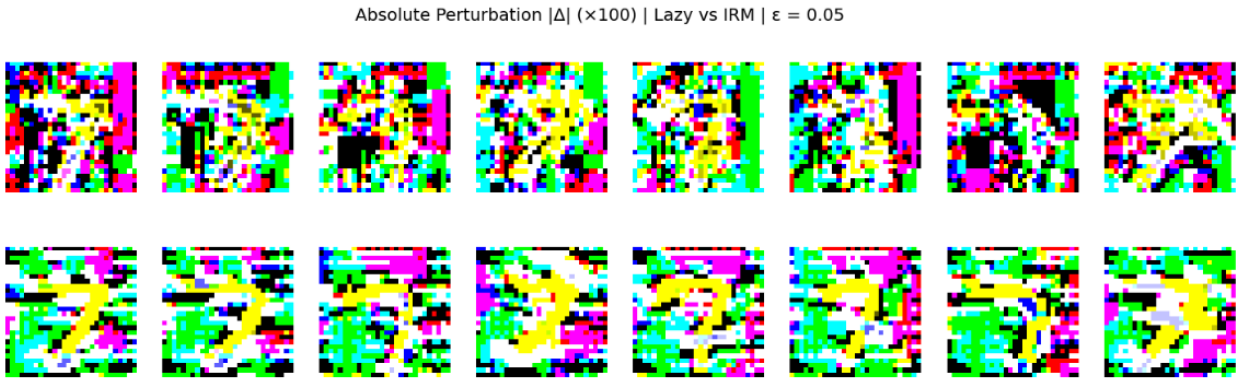
```
=== FGSM Results ===
Attack: 7 → 3
Untargeted | Lazy: 99.60% | Robust: 6.40%
Targeted   | Lazy: 13.00% | Robust: 2.00%
```

Untargeted attack means to push from & to any digit and targeted attack is to push from 7 to 3. Source and target can be dynamically changed in my script.



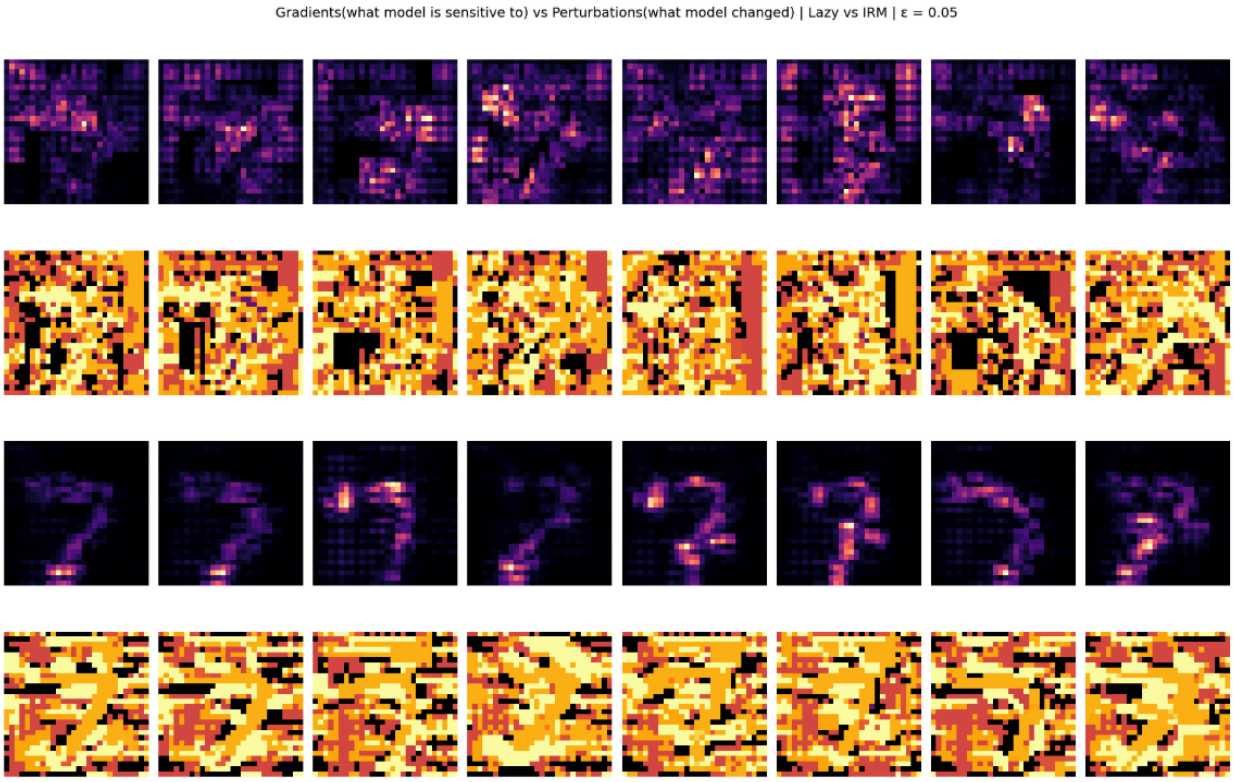
Top row - original Middle - modified for lazy Bottom = modified for IRM
There is no visible difference.

One recurring question was whether FGSM was truly finding adversarial structure in the model or just amplifying noise in the input. I also wondered how to interpret the perturbation visualizations. When I saw the “perturbation ×10” plots, I was unsure whether I was visualizing the new image, the signed perturbation, or just the magnitude of the change. This led to questions about whether removing or altering specific colors would still show up in the visualization, and whether taking the absolute value was hiding important directional information.

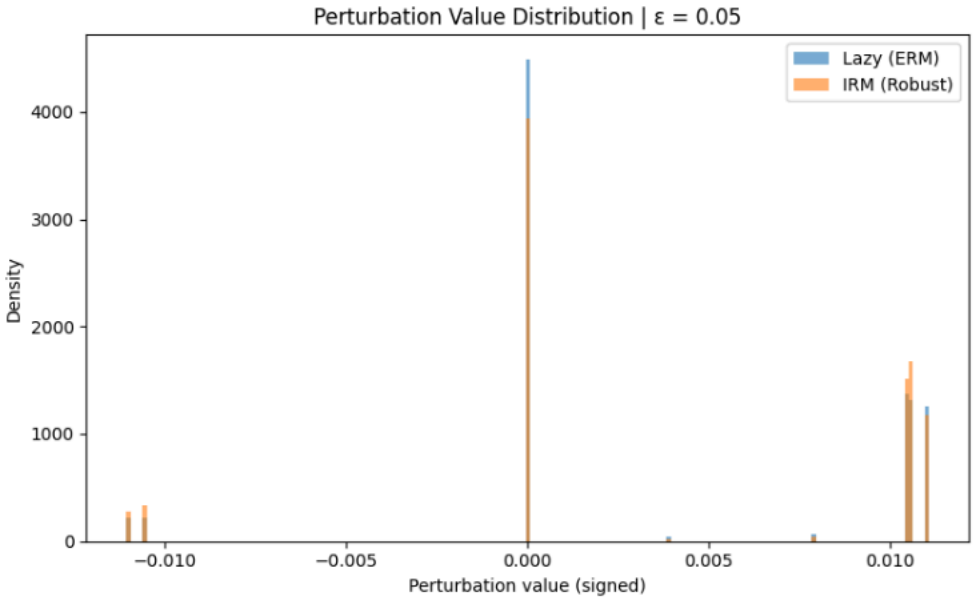


WE can see that for biased model pertubrations are kind of spread out and for fooling IRM they are more focussed. One preliminary indicator that LAzy model is easier to fool than IRM.

Gradcam to the rescue



This shows what the model is sensitive to and what changes were actually made. Clearly irm is more sensitive to shape than the colors which leads to cloudy noise like sensitivity pattern in lazy model.



The above chart show that statistically less changes were needed to fool the lazy model as compared to the IRM.

Upon reading this article https://medium.com/berkeleyischool/fgsm-attacks-on-mnist-fashion-dataset-90cd0eeed7ab I also became aware that FGSM's single linear step assumes local linearity of the model, which may not hold in regions of highly curved decision boundaries. This helped me understand why FGSM is often described as a baseline rather than a strong adversary. This explains why the accuracy for targeted attacks isnt so great despite hyperparameter finetuning.

Despite its limitations, FGSM turned out to be very useful for building intuition. The Lazy model was noticeably more sensitive to FGSM perturbations, often flipping predictions with relatively small epsilon values. The IRM model, on the other hand, appeared more stable, especially when epsilon was kept small. This contrast was one of the first indicators that the Lazy model might be relying on fragile, shortcut features such as color, while the IRM model was learning more invariant representations. FGSM made these differences visible quickly without requiring heavy computation.

Can we hide better ? PGD (Projected Gradient Descent)

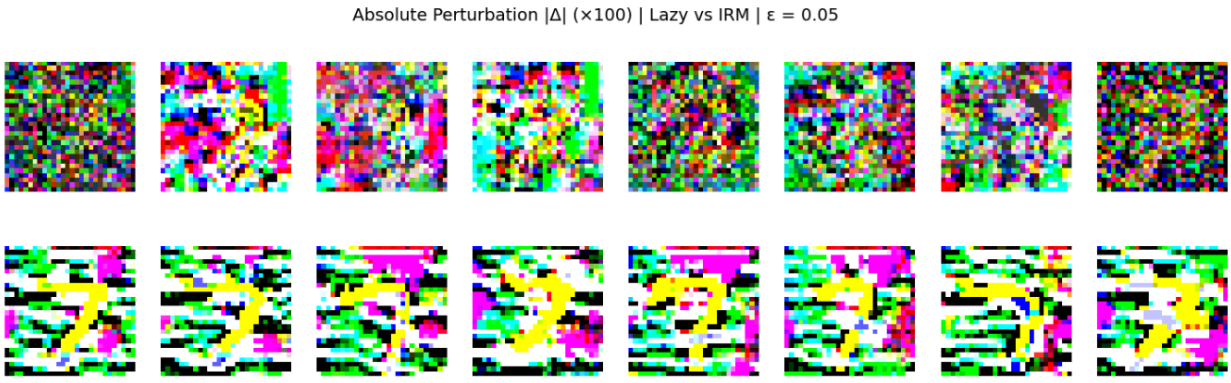
I wasnt satisfied with the accuracy of FGSM, I wanted something better. My intuition at this stage was to move beyond single-step perturbations and instead use something like a reverse Grad-CAM, where gradients are repeatedly used to mathematically optimize a noise pattern that actively pushes the model toward misclassification. This led me to implement Projected Gradient Descent. Unlike FGSM, which commits to a single gradient-sign step based on a local linear approximation, PGD treats adversarial example generation as an iterative optimization problem. At each step, the gradient is recomputed, the perturbation is refined, and the result is projected back into a fixed epsilon ball around the original image. This methodological difference is important because it allows PGD to adapt to curvature in the loss landscape and systematically search for worst-case perturbations, rather than relying on a single approximation.

The theoretical motivation for using PGD over FGSM became clearer both through reading and experimentation. FGSM can be viewed as one step of a constrained optimization, whereas PGD repeatedly solves this constrained problem, making it a much stronger first-order adversary. As argued by Madry et al.(7), robustness that does not hold against PGD is not meaningful in a worst-case sense. This theoretical distinction was reflected in the experiments as well. PGD consistently achieved higher attack success than FGSM under the same epsilon budget, exposing vulnerabilities that FGSM often missed. At the same time, the accuracy under PGD was still noticeably higher for the IRM model than for the Lazy model, reinforcing the insight that PGD is not only stronger but also more discriminative. It provided stronger evidence that the Lazy model relies on brittle shortcuts, while the IRM model has genuinely more robust decision boundaries.

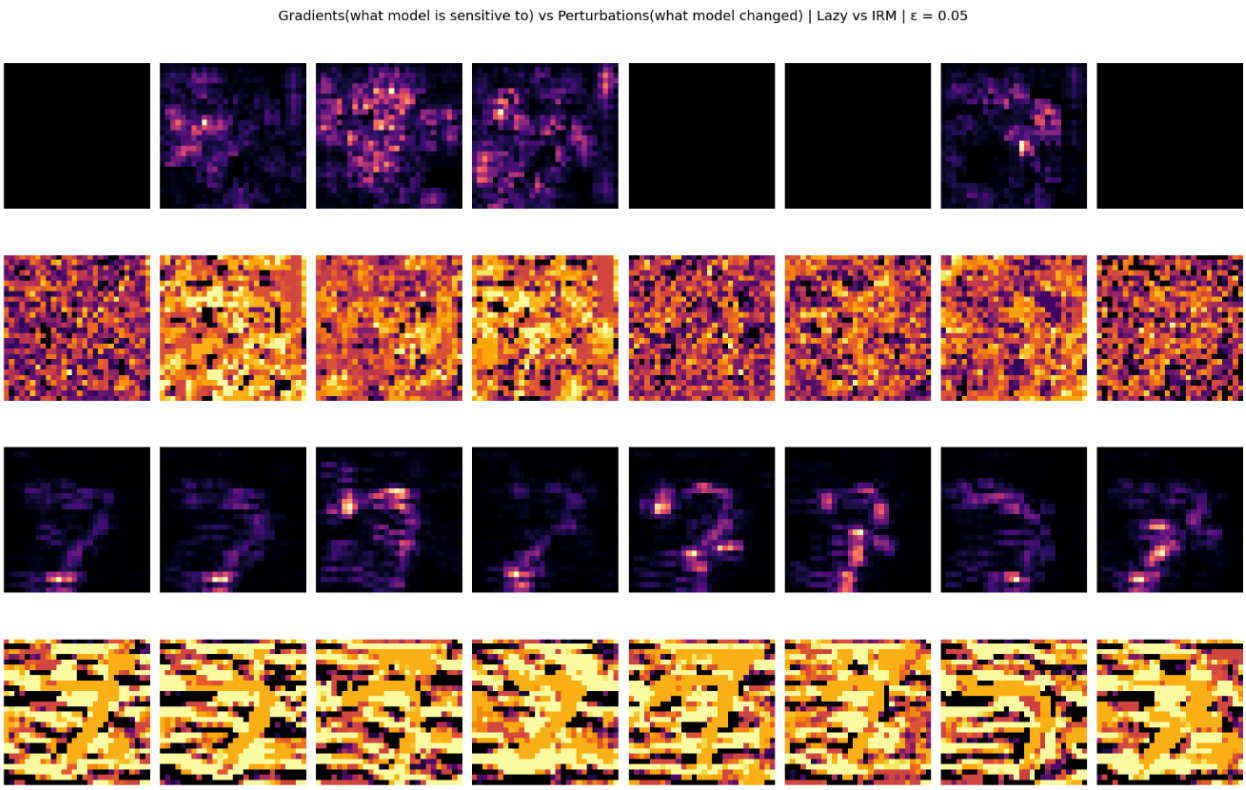
Much better than before

```
=== PGD Results ===
Attack: 7 → 3
Untargeted | Lazy: 100.00% | Robust: 6.00%
Targeted   | Lazy: 25.60% | Robust: 2.00%
```

The success rate against lazy model is almost twice as more than FGSM. Now it was time to see why ?



PGD perturbation clearly show that IRM model focuses on shape and thus to fool it you need to modify the pixels near the shape. Where as the lazy model can be fooled by messign up with background pixels as well, again indicating that the Lazy model is overreliant on shortcuts.



Trojan Horse of Attacks - Carlie and Winger

PGD was clearly stronger than FGSM, but even after seeing its results, I was not fully satisfied. The success rate, especially in the targeted setting, still felt like something I would hesitate to rely on if my chances of selection to Precog were actually at stake. That discomfort pushed me to look beyond first-order, gradient-sign based methods. While searching through papers and online discussions, I repeatedly came across the Carlini and Wagner attack (8) (https://medium.com/@zachariaharungeorge/adversarial-attacks-with-carlini-wagner-approach-8307daa9a503) , which was often described as one of the strongest and most reliable adversarial

attacks. This immediately stood out because it was framed not as a heuristic update rule like FGSM or PGD, but as a carefully designed optimization problem.

What fundamentally distinguishes the C&W method is how explicitly it formulates the adversarial objective. Instead of constraining the perturbation implicitly through projection, C&W directly optimizes for the smallest possible perturbation that causes misclassification. Mathematically, it minimizes the L2 norm of the perturbation while enforcing a margin-based misclassification constraint. This turns adversarial example generation into a continuous optimization problem, where the attack balances two competing goals: keeping the perturbation small and ensuring that the target class logit overtakes the original class logit by a desired confidence. The use of a differentiable surrogate loss and a change of variables, often via a tanh parameterization, ensures that the optimization remains smooth while still respecting input bounds.

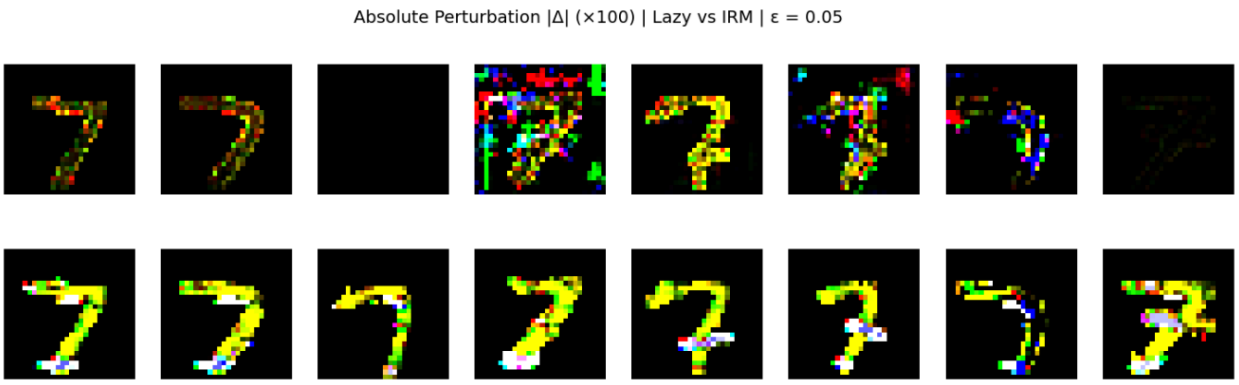
This formulation made it clear to me why C&W is considered more principled than both FGSM and PGD. FGSM relies on a single local linearization, and PGD improves upon this by iterating within a fixed epsilon ball, but both are still limited by a predefined threat budget. In contrast, C&W does not start by assuming an epsilon constraint at all. Instead, it asks a stronger question: what is the smallest possible change that forces the model to behave incorrectly. Because of this, C&W is able to find highly precise, low-norm perturbations that PGD may fail to discover, even with many iterations. In a sense, PGD explores a fixed region aggressively, while C&W reshapes the problem to explicitly search for the optimal point where the decision boundary is crossed.

The results literally made me very happy and curious

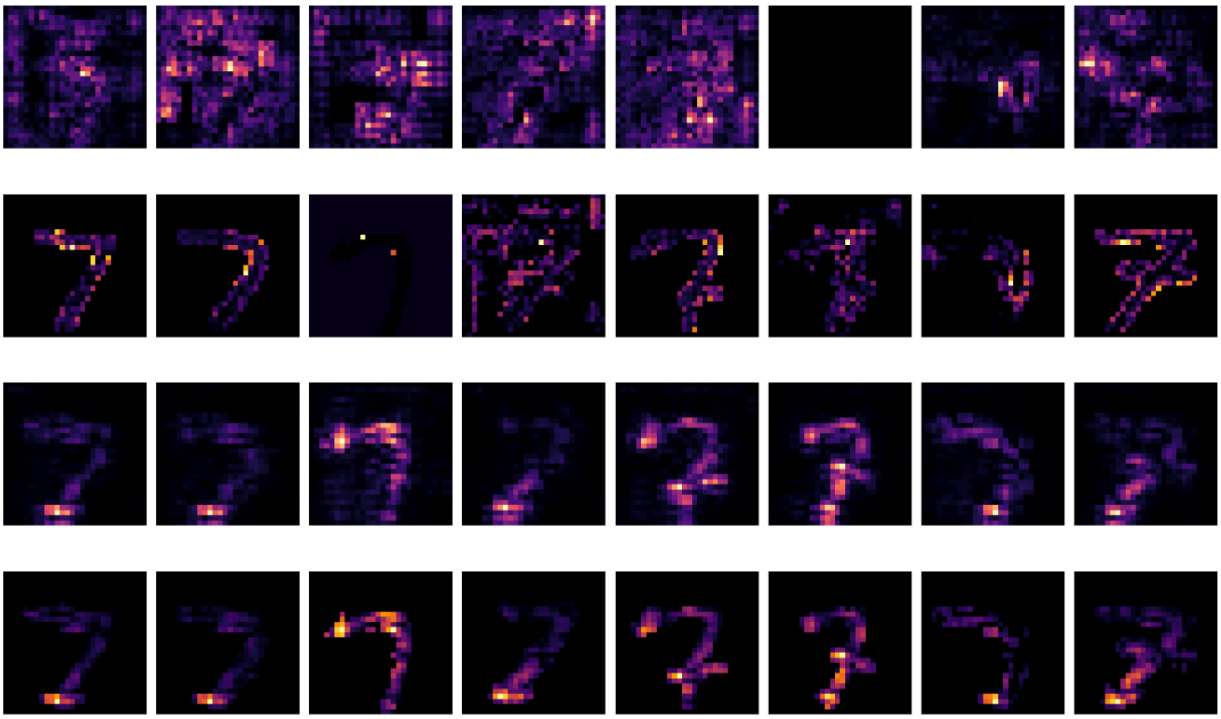
I was experimenting with these methods for kind of long time, when I ran the notebook, and saw the accuracy in front of me, I felt success.

```
=== C&W Results ===
Attack: 7 → 3
Untargeted | Lazy: 98.80% | Robust: 7.40%
Targeted   | Lazy: 40.60% | Robust: 2.20%
```

A whopping 40% accuracy against lazy model. IRM still wasnt budging anywhere.



The perturbations of CW were not at all like that of the earlier two methods.



Interestingly the perturbations for lazy and irm didnt have much difference despite having different gradient sensitivity distribution.

The Decomposition

My goal was to determine if I could decompose the messy, polysemantic hidden states of a color-biased model into cleaner, more interpretable features using Sparse Autoencoders (SAEs) (9). This was a journey of "trial and error" where I had to balance the L1 sparsity penalty to ensure the features were distinct enough to be meaningful without losing the ability to reconstruct the original data.

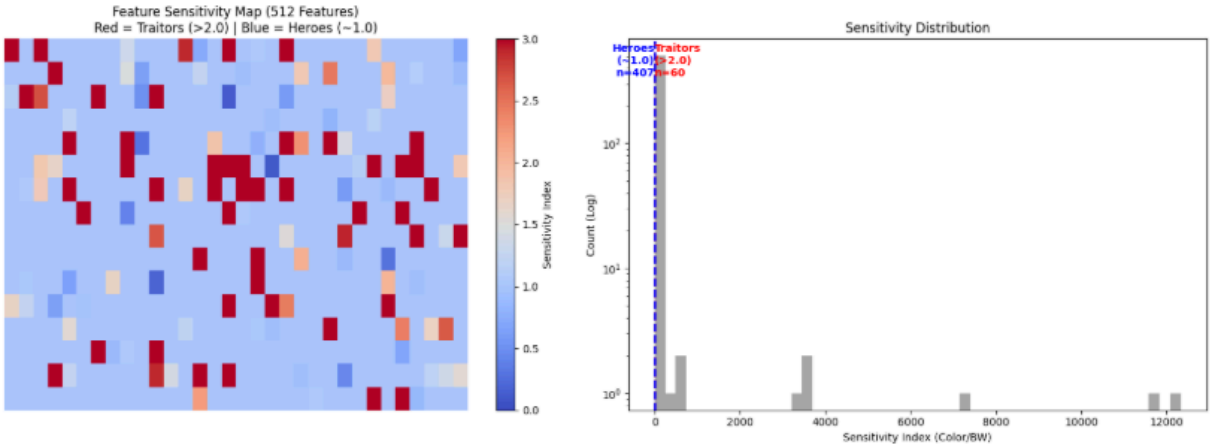
Implementing Sparse AutoEncoder

I built a script to act as a bridge between the model's raw activations and a more human-readable feature space. By hooking into the intermediate layers of the biased CNN, I extracted the hidden states and passed them through an encoder-decoder architecture designed for "overcomplete" representations. SAE is very popular in cv community. Here is my implementation -

1 Encoder and Decoder: The script projects the dense CNN activations into a larger feature dimension to untangle overlapping signals.
2 L1 Sparsity Penalty: I used a regularization term in the loss function to force the model to represent each input using as few active features as possible.
3 Reconstruction Accuracy: I monitored the MSE loss to ensure that the SAE could still faithfully reconstruct the original hidden state from its sparse code.

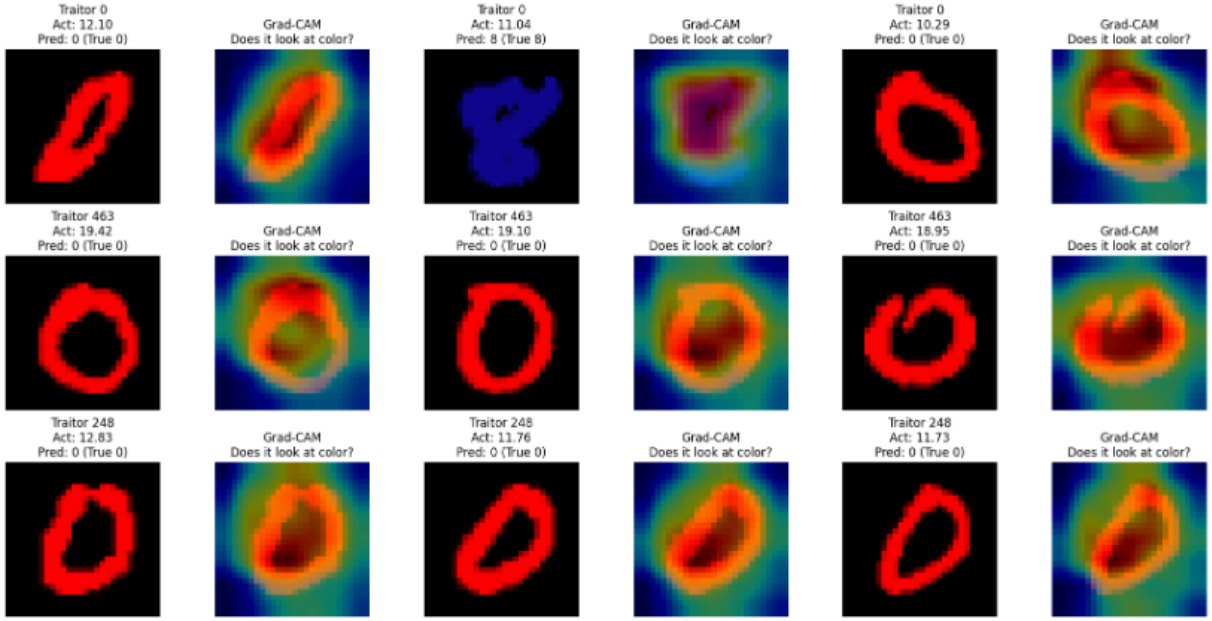
Identifying Heroes and Traitors

With the SAE trained, I could finally quantify the quality of the internal features. I used Mutual Information (MI) (Ratio between colour and shape sensitivity) as a primary metric to separate the neurons and SAE features into two distinct camps based on the information they encoded.

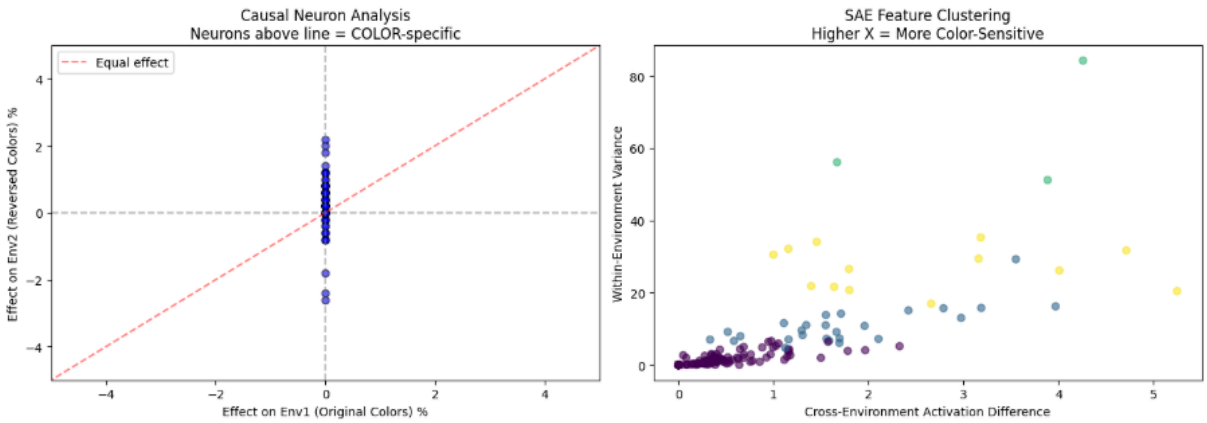


I see I learn, Visualising Features

To verify my MI findings, I implemented a visualize_neuron function that used my gradcam to synthesize the "ideal" input for specific features. I realized early on that raw optimization led to noisy artifacts, so I introduced spatial jitter and total variation regularization to stabilize the process. These visualizations revealed that "Traitors" were obsessed with color.

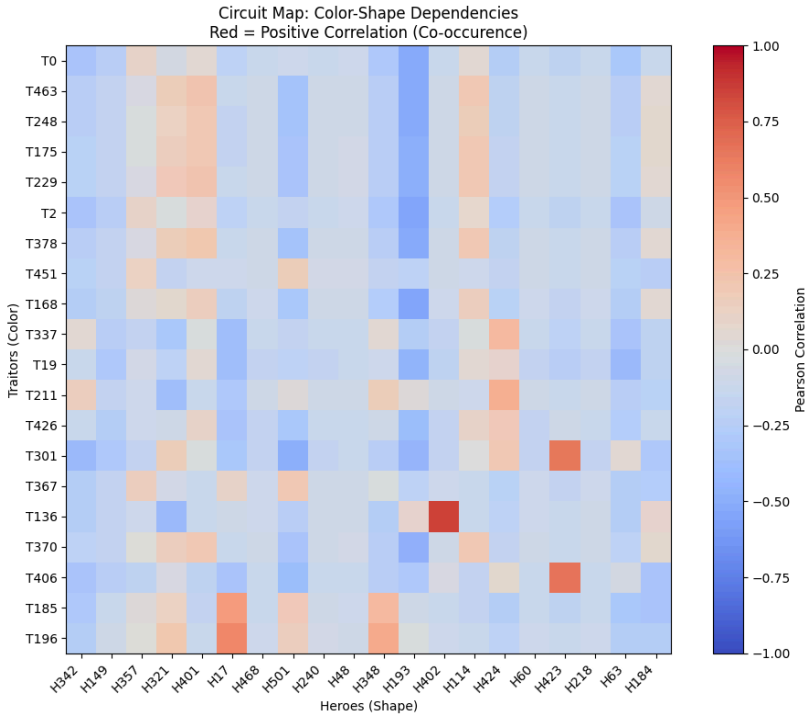


I tested features in biased dataset and bw dataset to measure their colour sensitivity



Causality vs. Correlation

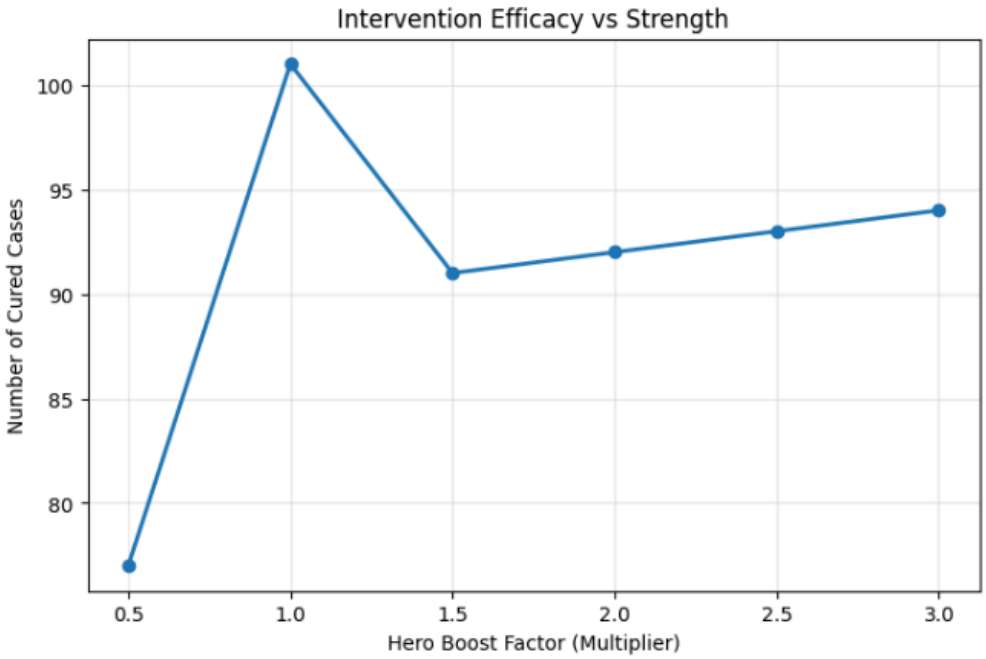
A major hurdle was the distinction between features that were just correlated with a class and those that were causal. I mapped out internal relationships to ensure my "surgery" targetted the right features.Logit Correlation: I looked at how the activation of a feature changed the final prediction scores for each digit. Identified Shortcuts: I found features highly correlated with a color that also had a strong positive correlation with a specific digit's logit, proving the model used the color as a direct shortcut.



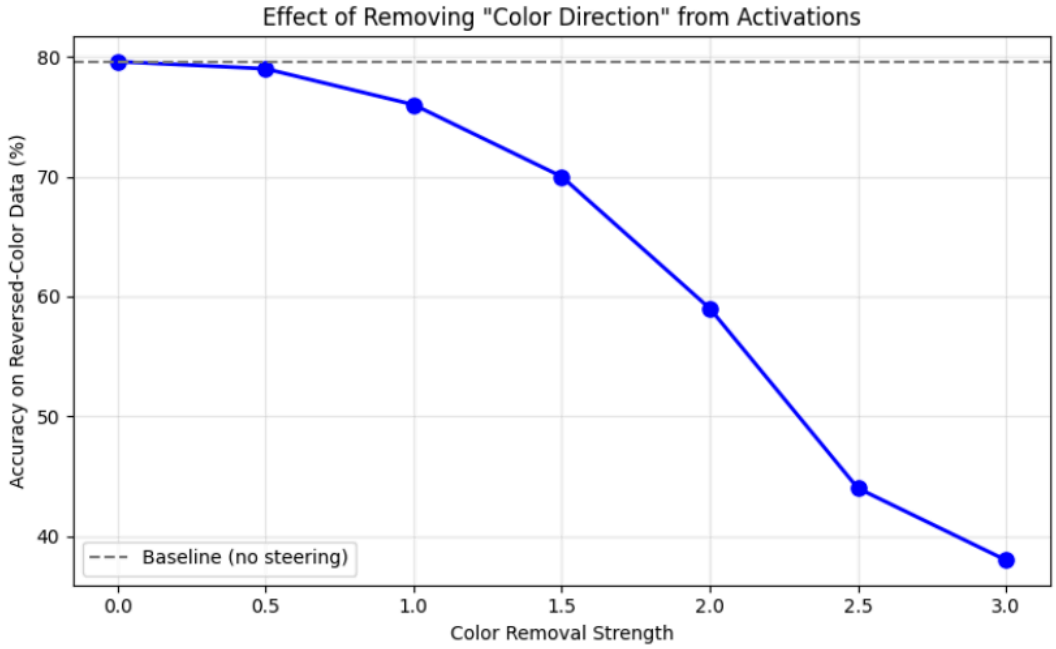
The Surgery (went well ?)

Once I knew which SAE features were the "Traitors," I moved from observation to intervention. I performed digital surgery on the model's hidden states to prove that these features were causal, not just correlated with the shortcuts.

Ablation (Cutting): I zeroed out the activations of color-specific SAE features. This caused a sharp drop in accuracy on the biased dataset, confirming the model's heavy reliance on shortcuts.



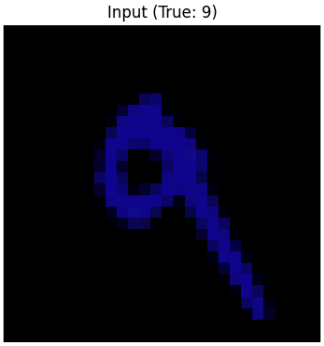
Steering (Amplifying): By artificially increasing the activation of a "Green" feature, I could trick the model into misclassifying a digit based purely on the internal "feeling" of that color.



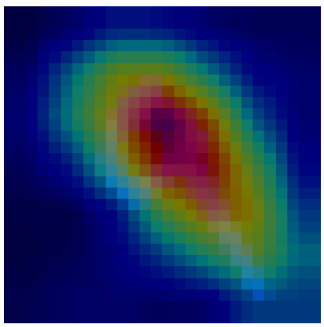
Grad Cam to the Rescue, again !

To see if the surgery actually changed the model's focus I used Grad-CAM to visualize where the network was looking before and after my interventions. Before the surgery, the Grad-CAM heatmaps for the biased model were often diffused across the background, showing that it was reading the color rather than the number. After I ablated the color features, the heatmaps shifted the model was forced to look at the actual digit contours to make its prediction. This confirmed that the SAE decomposition successfully identified the specific weights responsible for the shortcut behavior.

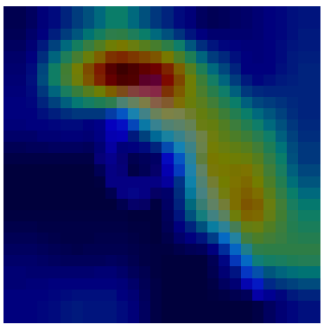
Input (True: 9)



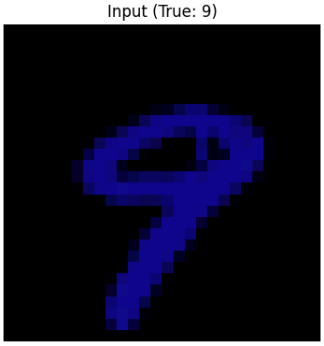
Original (Pred: 8)
Confusion



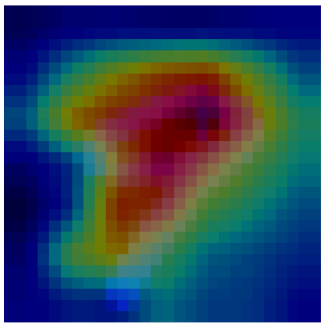
Cured (Pred: 9)
Focus



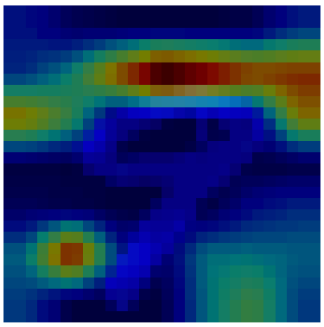
Input (True: 9)



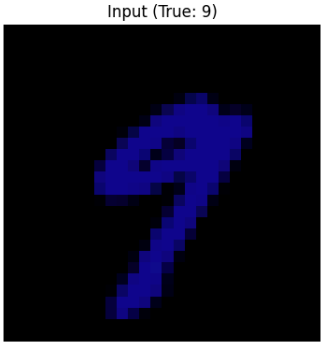
Original (Pred: 8)
Confusion



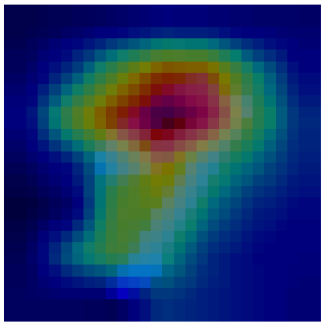
Cured (Pred: 9)
Focus



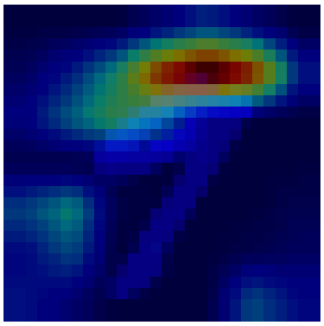
Input (True: 9)



Original (Pred: 8)
Confusion

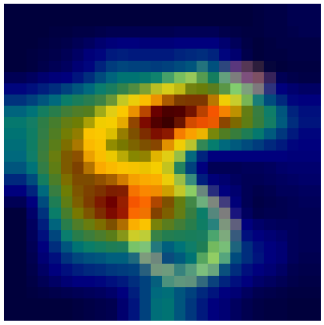


Cured (Pred: 9)
Focus

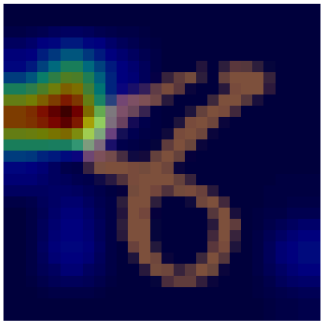


Input (True: 8)

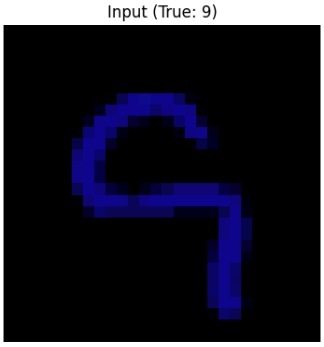
Original (Pred: 6)
Confusion



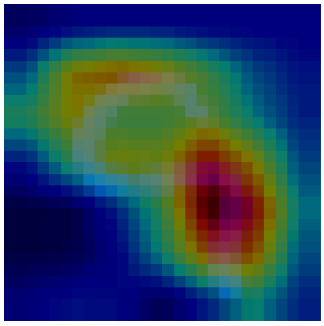
Cured (Pred: 8)
Focus



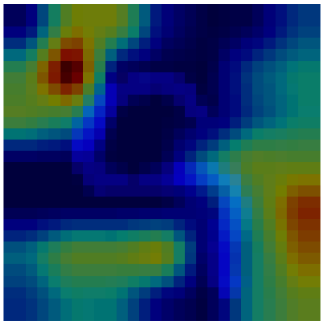
Input (True: 9)



Original (Pred: 8)
Confusion



Cured (Pred: 9)
Focus



Citations

1) Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791.

2) https://arxiv.org/abs/2209.10652 Elhage, N., Hume, T., Olah, C., Henighan, T., Hall, J., Carter, S., ... & Amodei, D. (2022). *Toy Models of Superposition*. Anthropic. arXiv:2209.10652.

3) Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., & Batra, D. (2017). Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*

4) https://medium.com/@stepanulyanin/implementing-grad-cam-in-pytorch-ea0937c31e82

5) Krueger, D., Caballero, E., Jacobsen, J. H., Zhang, A., Binas, J., Zhang, D., LePriol, R., & Courville, A. (2021).
Out-of-distribution generalization via risk extrapolation (REx).
Proceedings of the 38th International Conference on Machine Learning (ICML).
arXiv:2003.00688

6) Goodfellow, I. J., Shlens, J., & Szegedy, C. (2015).
Explaining and harnessing adversarial examples.
International Conference on Learning Representations (ICLR).
arXiv:1412.6572

7) Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018).
Towards deep learning models resistant to adversarial attacks.
International Conference on Learning Representations (ICLR).
arXiv:1706.06083

8) Carlini, N., & Wagner, D. (2017).
Towards evaluating the robustness of neural networks.
IEEE Symposium on Security and Privacy (SP).
arXiv:1608.04644

9) Makhzani, A., & Frey, B. (2013).
k-Sparse autoencoders.
arXiv preprint.
arXiv:1312.5663