# Precog Recruitment Task 2026
# **REPORT**

---

*Name:* *Sian Shinjo*
*Theme:* *Computer Vision*

---

## Introduction

This report details the work I've done for the Precog recruitment task under the theme of computer vision. It aims to explain the methodologies followed and interpret my findings and corresponding conclusions drawn over the course of undertaking this task.

As someone who's really interested in ML interpretability, computer vision techniques and adversarial robustness of models, the CV task represented a perfect amalgam of my interests, which motivated the choice of task for me.

## Tasks Completed

- Task 0: attempted, completed
  - *creation of a biased dataset (ColouredMNIST) with a spurious correlation to trick the model*
  - *colouring was done on the foreground stroke*
  - *two versions created: standard MNIST and FashionMNIST*
- Task 1: attempted, completed
  - *used ResNet-18 for the lazy model trained on the biased data*
- Task 2: attempted, completed
  - *optimised image pixels to maximise mean activation of channels; compared results with a control model*
  - *explored polysemanticity of neurons by visualising images with top activations for specific neurons*
- Task 3: attempted, completed
  - *implemented simplified GradCAM in PyTorch*
  - *used to generate heatmap analysis to visualise model focus*
- Task 4: attempted, completed
  - *implemented four strategies to train a robust model on the biased dataset without modifying training data*
- Task 5: attempted, completed
  - *ran targeted adversarial attacks on the lazy and robust models trained prior by introducing minimal perturbations*
- Task 6: attempted

## Task 0: The Biased Canvas

This task involved writing a script to synthetically generate a biased dataset to train our model on using the existing standard MNIST dataset of handwritten digits which is commonly used in computer vision projects.

The dataset that we were required to create had to have a specific spurious correlation aimed at biasing the model we would train on the dataset. The methodology I followed was to implement a custom wrapper class, ColouredMNIST, which inherits from PyTorch's standard Dataset library. This wrapper encapsulates the base dataset (referencing the 70,000 available images) and intercepts the data loading process to inject our specific color biases dynamically.

The 10 classes were mapped to a set of dominant colours chosen to be maximally distinguishable, to encourage lazy learning in the model.

**0: [1.0, 0.0, 0.0], # red**
**1: [0.0, 1.0, 0.0], # green**
**2: [0.0, 0.0, 1.0], # blue**
**3: [1.0, 1.0, 0.0], # yellow**
**4: [0.0, 1.0, 1.0], # cyan**
**5: [1.0, 0.0, 1.0], # magenta**
**6: [1.0, 0.5, 0.0], # orange**
**7: [0.0, 0.5, 0.0], # dark green**
**8: [0.5, 0.0, 1.0], # purple**
**9: [0.0, 0.5, 0.5], # teal**

The MNIST dataset already came pre-split into training (60,000 images) and testing (10,000 images) datasets. For the training set, 95% of images were assigned the dominant colour specified above as per the mapping, with the remaining 5% being assigned one of the colours at random. To evaluate reliance on this bias, for the test set, this relationship was inverted by **rotating the colour mapping by an offset of 5**, so that none of the test images were assigned the dominant colour specified above for their class.

The colouring methodology followed was to tint the **foreground strokes** without affecting the image background by using **tensor broadcasting**. The grayscale images (28x28) were expanded into 3-channel tensors, followed by an elementwise multiplication with the target colour vector. The black background (zero-valued) was unchanged, while the non-zero pixel values of the foreground were coloured by the transformation.

The provided 60,000 image set of training images was manually split at random into training (48,000 images) and validation (12,000 images) datasets, both of which maintained the 95% bias distribution.

*Note:* For this project, I implemented two versions – one using the standard MNIST dataset and another using the more complex FashionMNIST dataset. The switch to FashionMNIST was made because I was curious about the results I would obtain since it was noticeably more difficult to train models to recognise and distinguish classes in that dataset with high accuracy.

## Task 1: The Cheater

This task involved training a simple CNN on the biased training dataset and evaluating its performance on the validation and test datasets constructed.

I chose the standard ResNet-18 architecture for my model. But since the architecture was designed for ImageNet inputs (224x224) and our dataset consisted of 28x28 images, it would downsample the input very aggressively resulting in a loss of spatial information before the final layers. To fix this, these measures were taken:-

- The model was initialised with random weights (weights=None) to avoid prior shape biases.
- The initial 7x7 convolutional layer was replaced with a 3x3 version with stride 1 and padding 1 so as to work for small inputs.
- The initial maxpool layer was replaced with an identity layer to preserve feature map dimensions early in the network.
- The fully connected layer was modified for 10 output classes.

*[I found this resource really useful to understand how hyperparameters like kernel size, stride and padding affected transmission of information between layers: https://poloclub.github.io/cnn-explainer/]*

Initially, the model was trained over 5 epochs, with a batch size of 128 and the learning rate set to 0.0001. The loss function used was **torch.nn.CrossEntropyLoss** and the optimiser used was **Adam**.

This gave me an accuracy of 99.55% on the easy training set, but a very high 84.29% on the test set which was supposed to trick the model. This indicated that the model had correctly learned the features to identify the digits and the spurious correlation had been mostly ignored.

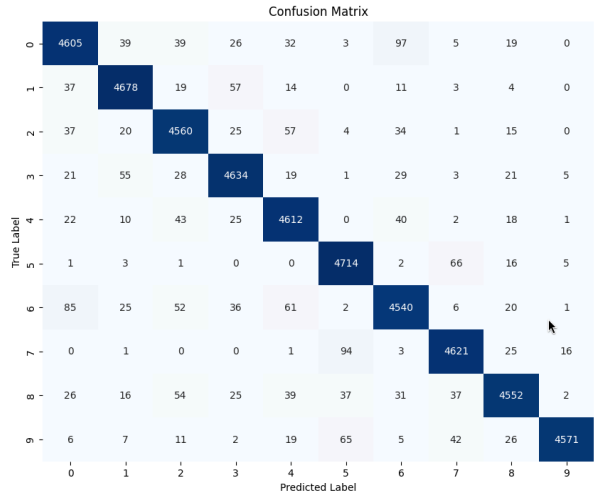
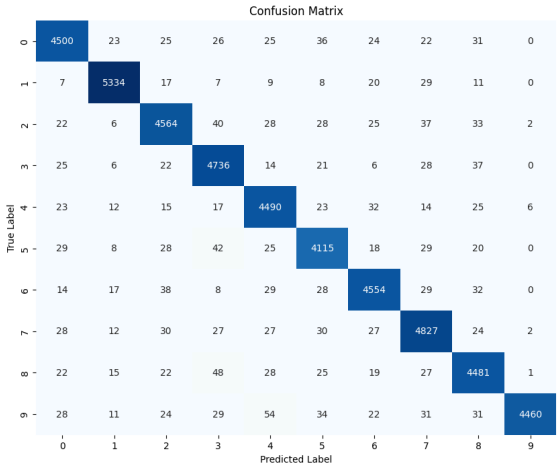
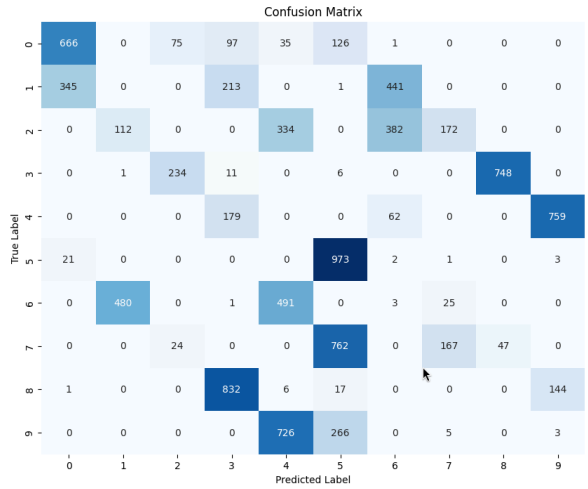
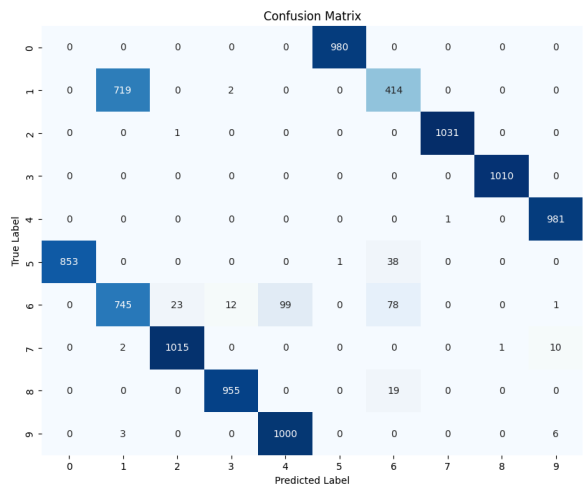
To try and get an accuracy under 20% for the test dataset (which would verify that the model had learned lazily), I tried different strategies:-

- I switched to the comparatively more complex dataset FashionMNIST, which had more difficult features to learn since it was a dataset of articles of clothing instead of just simple digits. Again, the foreground stroke was coloured and number of epochs for training (5) was unchanged. The greater complexity of the dataset was reflected in the calculated accuracy metrics: 98.11% on the training set, while the test set accuracy dropped to 74.43%.
- To try and decrease the accuracy further, I tried reducing the number of epochs during model training, so that it might not learn the correct features as well and might be deceived by the biased training data. Persisting with the FashionMNIST dataset and lowering the epoch count to 2, I observed 97.90% accuracy on the training set and 63.09% on the test set.
- Then I proceeded to experiment with the other hyperparameters. A factor 10 decrease in the learning rate to 0.00001 while keeping the epoch count at 2 yielded an accuracy of 97.01% with the training set and 46.39% with the testing set.
- Increasing the batch size to 256 led to a further drop in accuracy, possibly indicating overfitting of the model. Using an epoch count of 2 and a learning rate of 0.00001, the training set accuracy was still quite high at 96.14%, whereas the test set accuracy dropped to 22.34%, indicating that the attempt to bias the model was successful.

Further fine-tuning of the hyperparameters, with a batch size of 300, learning rate of 0.00001 and epoch count of 2 led to a high accuracy of **95.99% on the easy training set** and very low accuracy of **18.23% on the difficult test set**, indicating that we had successfully constructed a model that had overfitted the training data by learning the spurious correlation we engineered in the training dataset.

(These results were for the FashionMNIST dataset; the same implementation done later on standard MNIST gave an even lazier model that had 5-10% accuracy on the test set.)

The confusion matrices for the training and test data revealed that colour was the major feature that the model looked at when classifying images. For the FashionMNIST dataset, the confusion matrix for test data showed that for some numbers like (1, 5, 7) the model had tried to learn features other than colour. For the standard MNIST dataset, the bias was much stronger, and the model stuck to predictions based on the dominant colour for most numbers (except 1), as indicated by the shifted diagonal in the confusion matrix for test data.

| FashionMNIST | MNIST |
| --- | --- |
| Training:<br>  | Training:<br>  |
| Test:<br>  | Test:<br>  |

I validated this colour bias by taking a grayscale image of a certain class from either dataset and synthetically colouring it with the dominant colour of a target class. The model would then predict the target class based on this colouring, indicating that it had been deceived successfully.

## Task 2: The Prober

This task focused on interpreting the internal representations of our trained models. To understand *what* the neurons in our CNN were actually seeing, I simplified the procedure used by the OpenAI Microscope. The objective was to visualize the features that maximize

the activation of specific neurons, thereby revealing whether they were focusing on color, shape, or a combination of both.

**Experiment 1: Layer-Wise Visualisation**

To understand what maximally activates each neuron, I fixed the weights of the biased model and treated the input image itself as a set of learnable parameters, which could then be optimised. I implemented a function *visualise_neuron_layer* which performs the following steps:

1. **Initialization**: I started with an image tensor containing random noise sampled from a normal distribution. Crucially, I set *requires_grad=True* for this tensor, allowing PyTorch to calculate gradients with respect to the input pixels.
2. **Forward Hooking**: I utilised hooks to intercept the feature maps at a specific target layer.
3. **Optimisation Loop**: For a defined number of steps (300), I passed the noisy image through the model. I then calculated a loss function defined as the **negative mean activation** of the target neuron. By minimising this loss using the Adam optimiser (lr=0.05), the input image was iteratively updated to maximise the neuron's firing rate.
4. **Regularisation**: To ensure the generated images remained visually coherent and interpretable, I applied Gaussian blurring (kernel_size=3) every 4 steps and clamped the pixel values between 0 and 1.

This technique was then applied to conduct a scan across the layers of the ResNet-18 model, visualising 64 output channels (which correspond to neurons) at each layer. Since the ResNet-18 architecture consisted of layers with a variable number of output channels [64->128->256->512], only 64 channels were visualised at each layer (which could be set to the first 64 layers or chosen at random, as desired).

However, simply having the visualisations for the biased model didn’t help me draw meaningful conclusions. To understand how these visualisations would appear at each layer for a standard model that actually understood the features in the dataset, I decided to train a control model.

The control model was trained on the grayscale FashionMNIST dataset (the dataset chosen had to be unbiased so that the features would be learned properly by the model; I chose to avoid colouring the dataset because I wanted to see how the visualisation of neurons would appear if the model was unaffected by colour). The batch size, learning rate and number of epochs were kept the same as the biased model. This model achieved >80% accuracy on both the corresponding validation and

test datasets.
The layer-by-layer visualisation was done for both these models. It was first done in full colour to see if the colour bias was apparent at any layer. A grayscale version of the visualisation was also done for both models to reveal details about texture and shapes that might not be apparent to the human eye when in multicolour.

However, this visualisation only revealed limited insights:-

- The Layer 2 visualisation offered the most promising results, with clear, brightly-coloured filters appearing in the biased model's version. These filters were often almost monochromatic and prominently showed a dominant colour from the initial set.
- The grayscale visualisations showed more complex textures/patterns emerge in the control model's visualisation, indicative of detecting shapes/edges in the images.

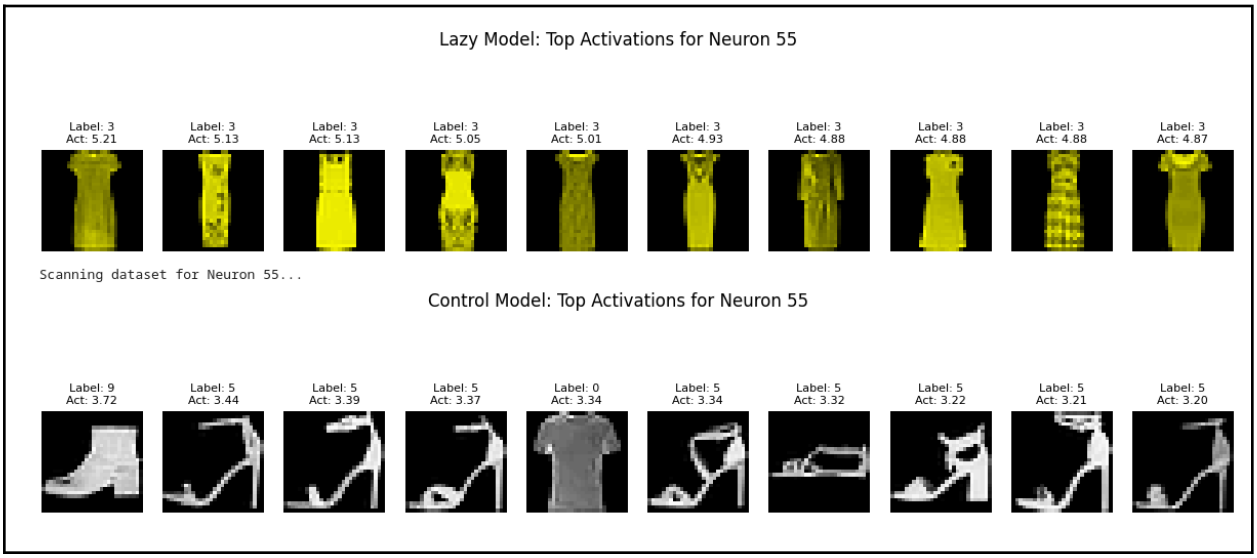
I hypothesised that this was due to the prevalence of **adversarial noise** due to the simple optimisation strategy used based on gradient descent. More advanced techniques like those used in the OpenAI Microscope rely on more robust regularisation techniques like Fourier parametrisation and transformation robustness to ensure more interpretable visualisations.

**Experiment 2: Investigating Polysemanticity via Top Activations**

To understand whether neurons might respond to multiple unrelated features simultaneously (i.e. exhibit a polysemantic nature), I implemented a function *get_top_activations* to identify the specific images in the dataset that maximally activated a given neuron.

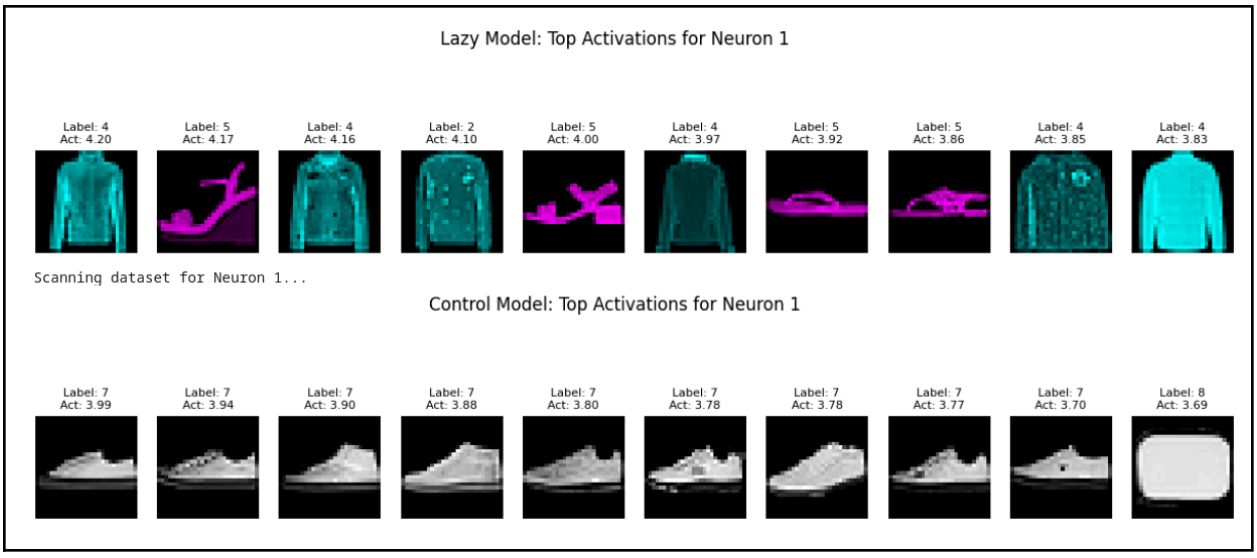
A random set of neurons was selected from the final convolutional layer (layer4) and the images having the top activations were analysed; this was done for the biased model in comparison with the control model.

First, images were taken from the validation dataset (which had the 95% bias). The biased model mostly selected images that shared the same colour, while the control model seemed to focus on structural features, which meant some objects belonging to different classes had similarly high activations.




Then, the same function was run on images taken from the test dataset, where the bias had been inverted. Here too the biased model tended to have neurons where the colour was the main common feature among the images with top activations, regardless of the classes being different this time.

Polysemanticity was greatly apparent through this, since some neurons seemed to specifically respond to two different classes of objects, and this was reflected in their top activations.



## Task 3: The Interrogation

This task sought to establish mathematically that the model was making lazy predictions by pinpointing what regions of the image were being looked at by the model when classifying them. To do this, I had to implement the GradCAM (Gradient-weighted Class Activation Mapping) library functions from scratch.

I created a custom GradCAM class that operates by hooking into the model's final convolutional layer (layer4). The process implemented followed these steps:-

1. **Forward Hook**: I registered a forward hook to capture the feature maps (A) produced by the final convolutional layer during a forward pass.
2. **Backward Hook**: I registered a backward hook to capture the gradients of the target class score ($y^c$) with respect to these feature maps ($\partial A/\partial y^c$).
3. **Global Pooling**: To determine the importance of each feature map (channel), I calculated a weight $\alpha_k^c$ by globally average-pooling the gradients. This mathematically quantifies "how much the k-th feature map contributes to class c."
4. **Weighted Combination**: I computed the final heatmap by taking the average of the weighted sum of the feature maps ($\sum k \alpha_k^c A^k$) and applying a ReLU activation. The ReLU is crucial because we are only interested in features that have a positive influence on the class of interest.
5. **Visualisation**: Finally, I upsampled this heatmap to the original image size (28×28) and overlaid it using the cv2.COLORMAP_JET colormap to visualize the "hotspots" of attention.

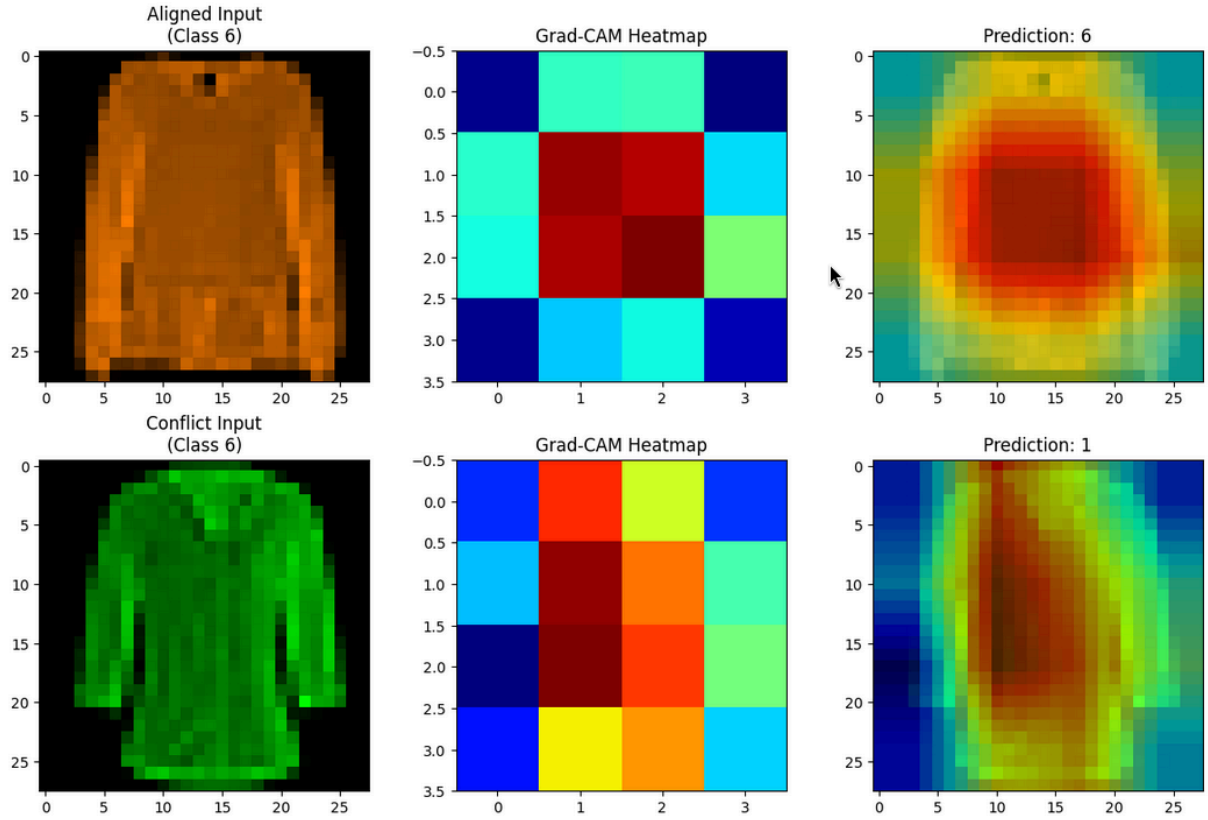
$$L^c_{\text{Grad-CAM}} = ReLU \underbrace{\left( \sum_k \alpha_k^c A^k \right)}_{\text{linear combination}}$$

The central formula behind GradCAM, as discussed in the paper 'Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization' by Selvaraju et al. [https://arxiv.org/pdf/1610.02391]

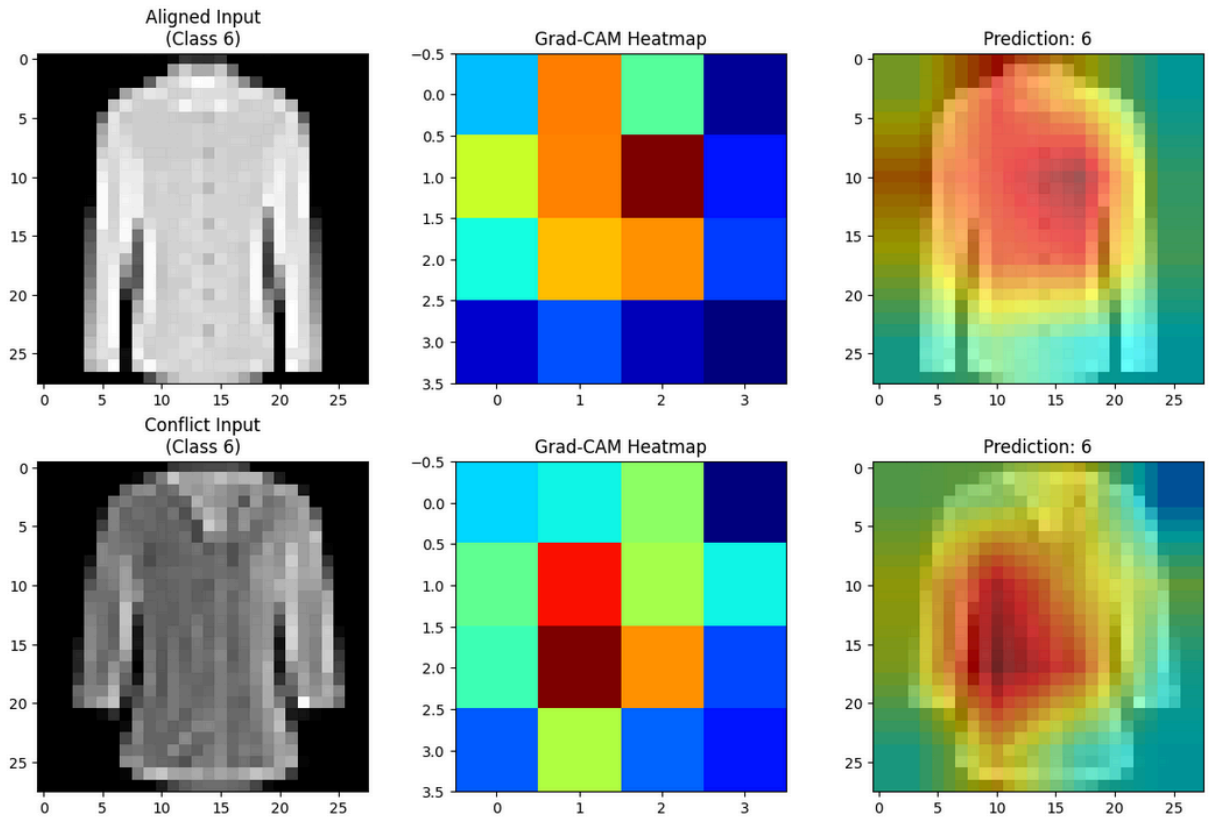
The heatmap analysis was again done in a comparative fashion for both the biased and control models. The heatmaps from the biased model were usually concentrated on the centre of the object (where the colour was most apparent), and avoided the edges or curves of the object. The heatmaps were obtained for both an aligned input image (satisfying the dominant colour bias) and a conflict input image (from the remaining 5% in the validation set, where the biased model was confidently wrong

in its prediction). Again, the reason for the incorrect prediction was apparent: the model focused on the areas of the object where the colour was most plainly visible.

In comparison, the heatmaps from the control model revealed a concentration on the sharp edges of the object and other distinguishable features (such as the large empty space near the straps in the case of the images of sandals from the FashionMNIST dataset). This corroborated the fact that the control model usually predicted the class accurately.




***Heatmaps from the biased model***
*Note that the heatmap avoids the edges of the image and is focused mainly over the coloured areas*




***Heatmaps from the control model***
*Note that the heatmap takes into account the edges of the image and the shape of the object*

## Task 4: The Intervention

Having proven the model's reliance on color, the next task was to retrain the model to ignore this bias and focus on shape without modifying the dataset (i.e., retaining the 95% color bias) or converting images to grayscale. To achieve this, I implemented a modular *Trainer* class that allowed me to switch between four distinct custom training strategies.

### Strategy 1: Importance Reweighting

**Implementation**: I implemented a reweighting mechanism where the standard Cross-Entropy loss is modified by a "focal term" $(1-p_t)^\gamma$. This dynamically assigns higher weights to examples where the model has low confidence.

**Hypothesis**: The biased training set contains a 95% majority of "easy" examples (colour matches label) and a 5% minority of "counter-examples" (random colour). A lazy model solves the majority easily but fails on the minority. By heavily penalising errors on the hard minority (the 5%), the model is mathematically forced to abandon the simple color rule and learn the robust feature (shape) that applies to all samples.

*(This was inspired by wondering why we had been asked to provide only a 95% bias to the dominant colour when creating the training dataset. The presence of conflict examples in the training data hinted that the model could be guided to focus on those when learning, to avoid giving importance to colour during classification.)*

**Strategy 2: Input Gradient Penalty**

**Implementation**: I added a regularisation term to the loss function based on the $L_2$ norm of the gradients of the loss with respect to the input images ($\lambda||\nabla_x L||^2$).

**Hypothesis**: Models that rely on spurious correlations (like specific color pixel values) often exhibit sharp, volatile decision boundaries – meaning a small change in pixel value causes a massive jump in gradients. By penalising large gradients, we force the model to learn a "smoother" function. Structural features (shape) tend to provide smoother gradients than high-frequency colour artifacts.

*(This is similar to R1 regularisation seen in training of Generative Adversarial Networks.)*

**Strategy 3: Edge-Prior Saliency Guidance**

**Implementation**: I engineered a custom loss function that guides the model to focus on the edges of the image as the most important features for classification. I used Sobel-Feldman kernels to extract the ground-truth edge map of the input images. During training, I calculated the model's saliency map (where the model is looking) and added a penalty term if the model focused on regions that were not edges (i.e., flat coloured areas).

**Hypothesis**: This was a direct intervention. If the model tries to minimise loss by looking at the flat red interior of a zero, the saliency penalty increases. To minimise both classification loss and saliency loss, the model must learn to focus its attention strictly on the contours of the digit.

*(This was inspired by looking at the Sobel kernels which when convolved over an image returns the edge map of the image. This was pretty cool to learn about and was interesting to adapt here.)*

**Strategy 4: Occlusion**

**Implementation**: I implemented a data augmentation strategy where a random 12×12 square patch (covering roughly 20% of the 28×28 image) is masked (set to black) before being fed into the model.

**Hypothesis**: Colour bias is often a local shortcut – seeing a single red pixel is enough to trigger a "Class 0" prediction. Shape, however, is a global, distributed feature. By randomly destroying parts of the image, we break the local colour shortcut. The model is forced to integrate information from the remaining visible parts to reconstruct the global structure, thereby learning robust shape features that survive partial destruction.

*(I had come across occlusion as a commonly-used technique when reading about convolutional neural networks here: https://cs231n.github.io/understanding-cnn/.)*

To test these strategies, I used the modularised *Trainer* class to train four separate models on the biased dataset over 15 epochs.

Overall, the most successful strategy for creating robust models was the reweighting strategy, which delivered consistently higher accuracy than the other strategies across both the MNIST and FashionMNIST datasets.

When using FashionMNIST, these models delivered an accuracy of 75-80% on the hard test dataset. For standard MNIST, the results were slightly better, with an average accuracy range of 85-90% on the hard test dataset.

## Task 5: The Invisible Cloak

This task explored the resilience of the models trained against Targeted Adversarial Attacks. The objective was to determine if the robust models (which learned shape) were harder to fool than the lazy model (which learned colour). Specifically, I attempted to perturb an image of a source class (e.g., a 7) such that the model would incorrectly predict a target class (e.g., a 3) with >90% confidence,

while keeping the perturbation invisible to the human eye (constraint: max pixel change $\epsilon \leq 0.05$).

The implementation was a version of Fast Gradient Sign Method (FGSM) written as an optimisation loop *targ_adv_attack* that had the following logic:-

1. **Gradient Calculation**: For each step, I calculated the gradient of the loss with respect to the input pixels to minimise the error with respect to the target class (not the true class). This was because my objective was to make the model predict the target class instead of the actual one.
2. **Pixel Update**: I updated all the pixels of the image in the direction of the gradient: $x_{new} = x - \alpha \cdot \text{sign}(\nabla_x L)$.
3. **Projection & Clamping**: To ensure the perturbation remained "invisible", I constrained the total change $\delta$ to fall within $[-\epsilon, +\epsilon]$ and clamped the final pixel values to $[0,1]$ to ensure validity of the image.
4. **Early Stopping**: The loop terminated immediately if the model's confidence in the target class exceeded 90%.

While the susceptibility of models to successful adversarial attacks (with epsilon under 0.05) was variable and depended on the relation between the input and target classes, it was observed that the **saliency guide-based model** was the most robust to such attacks, since the minimum value of epsilon required to fool this particular model was almost always over the specified threshold of 0.05. This could be explained due to the model's focus on edges in the image; structural changes to the image would require significant noise introduction, which wouldn't satisfy the epsilon threshold.

When comparing the lazy and robust models, it was apparent that the lazy model was sometimes just as difficult to fool while satisfying the epsilon threshold, since making it predict the target class would require changing the colour noticeably, which would amount to a significant noise introduction well above the epsilon threshold.

## Task 6: The Decomposition

I attempted to get a working implementation up for this task quickly to figure out how sparse autoencoders work. The existing sparse_autoencoders library at [https://github.com/ai-safety-foundation/sparse_autoencoder] was not installable on my system, so I opted for a simpler implementation guided by Gemini. The code gave me a huge collection of 4096 features and sets of images that satisfied those features, and I proceeded to visually inspect and select some features that I thought displayed

certain patterns. These are documented in the MNIST python notebook. However, on trying to boost certain features to make the model change its predicted class to a target class, the results obtained were quite confusing and required further inspection. They did not seem to agree with the patterns I had noticed or expected in the feature sets, and could not be explained easily.

Unfortunately, though I saved the SAE model weights to allow reproducibility, the SAE data was not saved, so on rerunning the notebook, the features indices I had selected no longer reflected meaningful features, so I was unable to continue working on it due to time constraints. However, my approach and attempts have been left in the python notebook for reference.

---

**References:**

Gemini was used for a lot of the research and implementational doubts that cropped up over the course of the project, since I was new to PyTorch and was still learning about convolutional neural networks.

Other references:-


- https://poloclub.github.io/cnn-explainer/
- https://colab.research.google.com/github/reiinakano/invariant-risk-minimization/blob/master/invariant_risk_minimization_colored_mnist.ipynb#scrollTo=knP-xNzavgAb
- https://docs.pytorch.org/tutorials/beginner/basics/data_tutorial.html
- https://docs.pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
- https://medium.com/@myringoleMLGOD/simple-convolutional-neural-network-cnn-for-dummies-in-pytorch-a-step-by-step-guide-6f4109f6df80
- https://cs231n.github.io/convolutional-networks/
- https://cs231n.github.io/understanding-cnn/
- https://cs231n.github.io/transfer-learning/
- https://web.stanford.edu/~nanbhas/blog/forward-hooks-pytorch/
- https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm
- https://arxiv.org/pdf/1610.02391