Towards Global Neural Network Feature Importance for Image Data

Names of Group Members

Member 1 – Hetarth Chopra (hetarth2)

Member 2 – Avinoor Singh Kohli (askohli2)

Member 3 – Tarun Anoop Sharma (tsharma7)

Changes from the Proposal –

While working on the project, we realized that the algorithms can be implemented, however from the feedback we realized that we could add more algorithms as well as use two (and more datasets/end-goals (for image generation) which we will show in the final submission) to the given idea. Other than using Accumulated Gradients Approach and Integrated Gradients, which are seen below, we are also working on implementing GradCam, which is another approach for calculating local feature importance. (https://jacobgil.github.io/pytorch-gradcam-book/introduction.html), specifically we will use EigenCam.

All four approaches are now being evaluated on the same datasets – FashionMNIST and SVHN.

Summary of the work done –

The main progress update is divided into three parts

a – Accumulated Gradients –

The algorithm works in the following way –

Let's denote:

- X is the input tensor
- $f(X)$ is the model's prediction function for the input X
- $\frac{\partial f(X)}{\partial X}$ as the gradient of the model's predictions with respect to the input X.
- $(|\cdot|)$ as the absolute value function.
- $(\text{mean}_0(\cdot))$ as the operation of taking the mean across the 0-th axis (or a specified axis).

The equation for the update of the feature importance in a single iteration can be represented as:

$$\text{feat\_imp} += \left|\text{mean}_0\left(\frac{\partial f(X)}{\partial X}\right)\right|$$

With this implemented, a typical PyTorch loop will look like this->

```
    loop = tqdm(enumerate(data_loader), total=len(data_loader), leave=False)
    for batch, (X, y) in loop:
        X, y = X.to(device), y.to(device)
        if calculate_feat_imp and X.requires_grad == False:
            X.requires_grad = True
        y_pred = model(X)
        loss = loss_fn(y_pred, y)
        if calculate_feat_imp:
            with torch.no_grad():
                gradients = torch.autograd.grad(outputs=y_pred, inputs=X, grad_outputs=torch.ones_like(y_pred),
only_inputs=True, retain_graph=True)[0]
```

```
                feat_imp += torch.abs(gradients).mean(dim=0)
            train_loss += loss.item()
            train_acc += accuracy_fn(y, y_pred.argmax(dim=1))
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            # Update progress bar
            loop.set_description(f"Epoch [{epoch+1}/{epochs}]")
            loop.set_postfix(loss=loss.item(), acc=accuracy_fn(y, y_pred.argmax(dim=1)))
        if calculate_feat_imp:
            feat_imp /= len(data_loader)
            feat_imp = feat_imp.cpu().numpy()
        train_loss /= len(data_loader)
        train_acc /= len(data_loader)
        print(f"\nTrain Loss: {train_loss:.5f} | Train Acc: {train_acc:.2f}%")
        return feat_imp if calculate_feat_imp else None
```

The coded algorithm has been highlighted in blue. This code theoretically allows a user to accumulate the gradient change of the output (y_pred), per feature, per epoch with respect to the input image (X). It returns a tensor feat_imp, which is of the same shape and size as a sample image from the given dataset. We decided it to test with the FashionMNIST dataset as well as for the SVHN dataset, using it on both on a pretrained CNN model and a normal CNN model. The model definitions, for FashionMNIST, are given as below –

```
class FashionMNISTModel(nn.Module):
    def __init__(self, input_shape: int,hidden_units: int, output_shape: int) -> None:
        super().__init__()
        self.conv_block_1 = nn.Sequential(
         nn.Conv2d(in_channels = input_shape ,out_channels = hidden_units,kernel_size=3,stride=1,padding=1),
         nn.ReLU(),
         nn.Conv2d(in_channels =hidden_units, out_channels = hidden_units, kernel_size=3, stride= 1, padding=1),
         nn.ReLU(),
         nn.MaxPool2d(kernel_size=2))
        self.conv_block_2 = nn.Sequential(
         nn.Conv2d(in_channels = hidden_units ,out_channels = hidden_units,kernel_size=3,stride=1,padding=1),
         nn.ReLU(),
         nn.Conv2d(in_channels =hidden_units,out_channels = hidden_units,kernel_size=3,stride= 1,padding=1),
         nn.ReLU(),
         nn.MaxPool2d(kernel_size=2))
        self.classifier = nn.Sequential(
            nn.Flatten(),
            nn.Linear(in_features=hidden_units*7*7,out_features=output_shape))
    def forward(self,x):
        x = self.conv_block_1(x)
        x = self.conv_block_2(x)
        x = self.classifier(x)
        return x
class FashionMNISTPretrained(nn.Module):
    def __init__(self, output_shape: int) -> None:
        super().__init__()
        self.resnet18 = models.resnet18(pretrained=True)
        num_ftrs = self.resnet18.fc.in_features
        self.resnet18.fc = nn.Linear(num_ftrs, output_shape)
```

On training this model with the SGD Optimizer and Cross Entropy Loss for 20 epochs and a batch size of 256, we were able to arrive at a feature importance matrix that looks like in the Figure 1.

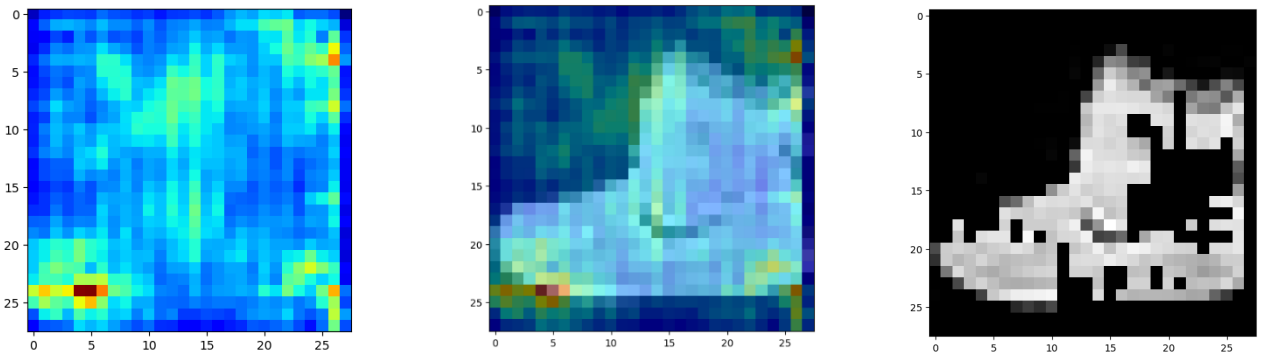

Figure 1 Feature Importance Matrix and it's overlay on a single image, for a FashionMNIST Model trained on 30 epochs. The red in this image point to higher magnitude for feature importance and blue represents areas of low feature importance. The third image retains the top 60% of the image and reducing the rest of the features as "0".

The first experiment that we performed was to prune the "image" and create a global mask of the top N% of the important features in the image (leading to N% compression in the dataset), and then the compressed dataset is trained using same hyperparameters, and the drop in accuracy (on the test dataset) is noted. The results can be seen as below –

Model Name	Batch Size	Optimizer (with LR)	Accuracy on 20 epochs	Top N% of features	Accuracy on training Pruned Images	Drop in Accuracy
FashionMNIST-Normal	256	Adam (LR-0.001)	88.68%	60%	85.94%	2.74%
FashionMNIST-Normal	256	Adam (LR-0.001)	88.68%	40%	84.43%	4.25%
FashionMNIST-Normal	256	Adam (LR-0.001)	88.68%	10%	78.54%	10.14%
SVHN-Normal	256	Adam (LR-0.001)	87.44%	60%	87.01%	0.43%
SVHN-Normal	256	Adam (LR-0.001)	87.44%	40%	86.66%	0.78%
SVHN-Normal	256	Adam (LR-0.001)	87.44%	10%	81.61%	5.83%
FashionMNIST-PretrainedResnet18	256	Adam (LR-0.001)	90.47%	60%	88.39%	2.08%
FashionMNIST-PretrainedResnet18	256	Adam (LR-0.001)	90.47%	40%	86.80%	3.67%
FashionMNIST-PretrainedResnet18	256	Adam (LR-0.001)	90.47%	10%	81.24%	9.23%
SVHN-PretrainedResnet18	256	Adam (LR – 0.001)	89.58%	60%	88.15%	1.43%
SVHN-PretrainedResnet18	256	Adam (LR – 0.001)	89.58%	40%	87.82%	1.76%
SVHN-PretrainedResnet18	256	Adam (LR – 0.001)	89.58%	10%	82.33%	7.52%

For the FashionMNIST dataset, using the normal model and varying N%, accuracy after training on pruned images dropped by 2.74% at 60% features, 4.25% at 40% features, and significantly by 10.14% at 10%

features, from a baseline of 88.68%. In contrast, the SVHN dataset showed a lesser impact on accuracy, with reductions of 0.43%, 0.78%, and 5.83% respectively, from a baseline of 87.44%. Notably, when utilizing a pretrained ResNet18 model, the FashionMNIST dataset showed smaller drops in accuracy of 2.08%, 3.67%, and 9.23% at the same respective feature levels, starting from a higher baseline of 90.47%. Meanwhile, the SVHN dataset with the pretrained ResNet18 showed even smaller declines in accuracy of 1.43%, 1.76%, and 7.52% from an 89.58% baseline. These results indicate that the degree of feature pruning has a consistent impact on model performance, with higher feature retention generally leading to less accuracy degradation, and the effects varying by model complexity and dataset characteristics.

The second experiment is to visualize the Feature Importance Matrix during the training, and study it. We create a video of all image instances of Feature Importance throughout the training. A couple of frames are shown below in Figure 2. On visually inspecting the video, one can see that after a few epochs the global feature importance maps do not change a lot (as can be inferred from Figure 2 as well). It seems like a good idea to pursue studying it's variance as the epochs progress. It is also seen that batch size also plays a significant role in the variance of the global feature importance.

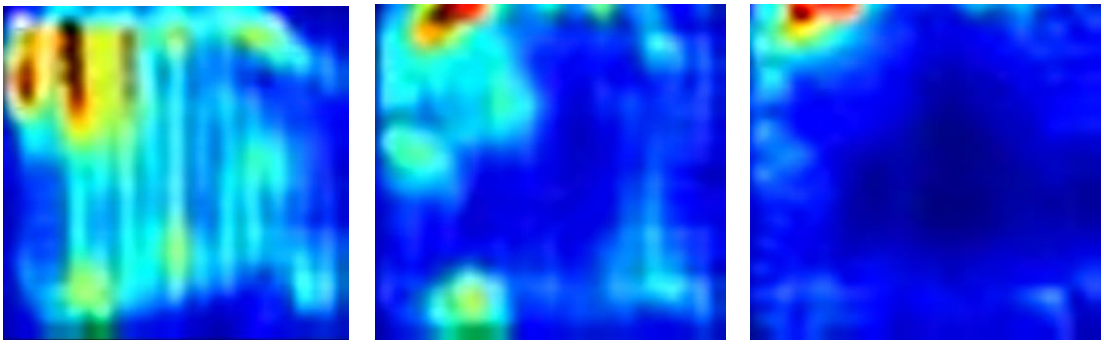

Figure 2 Evolution of the Feature Importances over epochs. The first image is immediately after 1 epoch, the second is after 10 epochs, and the third is after 20 epochs. Videos are attached in the zip file.

The third experiment we did was to find out local feature importances using the same methodology. To do this, we first train the model on a given dataset, after that, we use the same pretrained weights and apply the same logic to just a single image going through the forward pass. This helps us to find out, from an

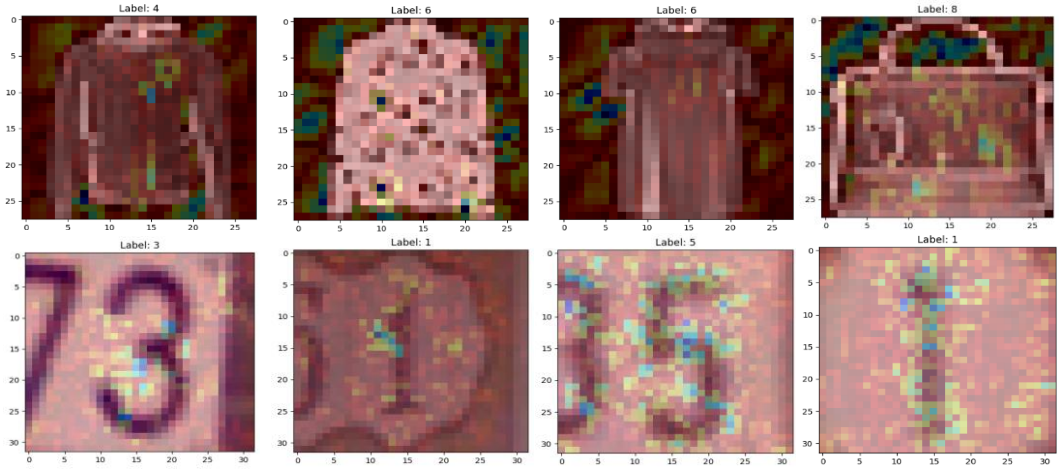

Figure 3 Local Feature Importances captured for selected samples from Fashion-MNIST and SVHN Dataset. We can see that for both colored and grayscale images, we are able to point out important features on both datasets which are unique to each image.

already trained model, which are the important activations of an image with respect to its gradient. Some visualizations can be seen from the below Figure 3.

B – EigenCam –

Eigen-CAM, as stated in the study, is a visualization tool that helps us understand how convolutional neural networks (CNNs) learn from data and why they perform poorly or well on specific tasks. This method builds on the concept of Class Activation Maps (CAM), but it takes a more obvious approach by utilizing the principal components of learned features from the network's convolutional layers. Essentially, Eigen-CAM computes and visualizes the main components (the most significant, defining properties recognized by the network) of the activation maps generated by the convolutional layers. Eigen-CAM operates directly on convolution layer outputs and does not require any changes to the CNN's architecture or model retraining. The algorithm is highlighted in blue.

```
class EigenCAM(CAM):
    def get_heatmap(self, img):
        img_rgb = img.convert('RGB') if isinstance(img, Image.Image) else Image.fromarray(img).convert('RGB')
        tensor = self.prep(img_rgb)[None, ...].to(self.device)
        output = self.model(tensor)
        feature = self.feature['output']
        self._check(feature)
        _, _, vT = torch.linalg.svd(feature)
        v1 = vT[:, :, 0, :][..., None, :]
        cam = feature @ v1.repeat(1, 1, v1.shape[3], 1)
        cam = cam.sum(1)
        cam = cam - cam.min()
        cam = cam / (cam.max() - cam.min())
        cam = cam.detach().cpu().numpy().squeeze(0)
        cam = cv2.resize(cam, img_rgb.size)
        cam = np.uint8(255 * cam)
        heatmap = cv2.applyColorMap(cam, cv2.COLORMAP_JET)
        img_array = np.array(img_rgb)
        overlay = cv2.addWeighted(img_array, 0.7, heatmap, 0.3, 0)
        return output, overlay
```

We use both the datasets with EigenCAM on FashionMNIST and we can see some outputs as seen in the Figure 4

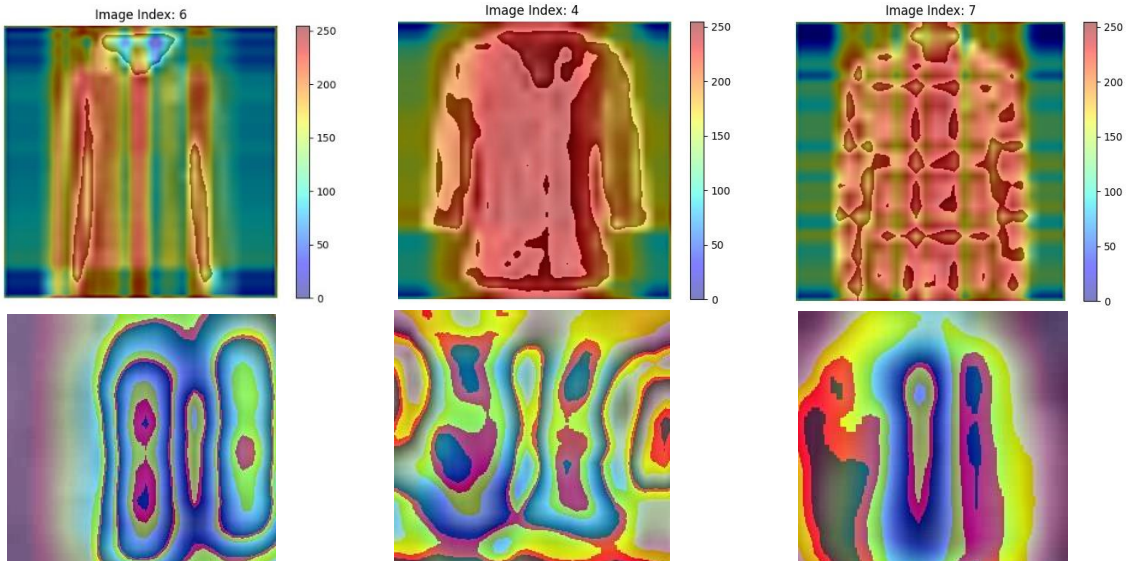

Figure 4 Outputs of EigenCAM on FashionMNIST (above) and SVNH dataset (below)

C – Integrated Gradients -

The Integrated Gradients method, as discussed in [2], is guided by two fundamental axioms—Sensitivity and Implementation Invariance—aiming to overcome limitations in existing attribution methods that often do not satisfy these principles.

The algorithm works in the following way:

- Baseline and Path Definition: The algorithm requires a baseline input (typically representing an 'absence' of features) and considers the straight-line path in the input space from this baseline to the actual input.
- Gradient Calculation Along Path: For each point along the path, the gradients of the network's output with respect to the input features are computed.
- Integration of Gradients: These gradients are then integrated along the path from the baseline to the input. This integration is essentially the accumulation of gradients and provides the contribution of each input feature to the output prediction.

The equation for this is given as:

$$IntegratedGrads_i(x) = (x_i - x_i') \times \int_{\alpha=0}^{1} \frac{\partial F\left(x' + \alpha \times (x - x')\right)}{\partial x_i} d\alpha$$

Where:

- F represents the model.
- x_i and x_i' are the i[th] components of the input and baseline, respectively.
- $\frac{\partial f(x)}{\partial x_i}$ as the gradient of $F(x)$ along the i[th] dimension.
- α scales the path from the baseline to the input.

This can be implemented in pytorch as follows:

```
def integrated_gradients(input, referenceImage, net, label):
    gradAvg = 0
    estIterations = 100
    for i in range(estIterations):
        x = referenceImage
        x = x + ((i+1) / estIterations) * (input - referenceImage)
        x.requires_grad = True
        output = net(x)
        predictProbab = output[0, label]
        grad = torch.autograd.grad(predictProbab, x)[0]
        gradAvg += grad / estIterations
    gradients = gradAvg * (input - referenceImage)
    return gradients
```

We created two separate implementations of these, considering two different use cases:

- Computing gradients on a custom model using the test set.
- Computing gradients on a custom dataset on any pretrained model.

For computing gradients on a custom model using the test set:

We trained a custom model on MNIST dataset and computed gradients on the corresponding test set. The relevant code is as follows:

```
referenceImage = np.zeros((28, 28, 1), dtype=np.float32)
referenceImageProc = transform_test(referenceImage).unsqueeze(0)
index = 0
for image, label in igloader:
  image = image.to(device)
  label = label.to(device)
  gradients = integrated_gradients(image, referenceImageProc, net, label)
  gradients = gradients.squeeze(0)
  gradients = gradients.permute(1, 2, 0)
  imagerecon = image.squeeze(0)
  imagerecon = transforms.Normalize((0,), (1/0.5,))(imagerecon)
  imagerecon = transforms.Normalize((-0.5,), (1,))(imagerecon)
  imagerecon.permute(1, 2, 0)
  imagerecon = imagerecon.squeeze(0)

  gradientsCpu = np.fabs(gradients.cpu().data.numpy())
  gradientsCpu = gradientsCpu / np.max(gradientsCpu)

  images = [imagerecon.cpu().data.numpy(), gradientsCpu]

  save_and_show_output(images, classes[label], "output/test"+ str(index) + ".jpeg")
```

A few sample outputs of the code are shown in Figure 5

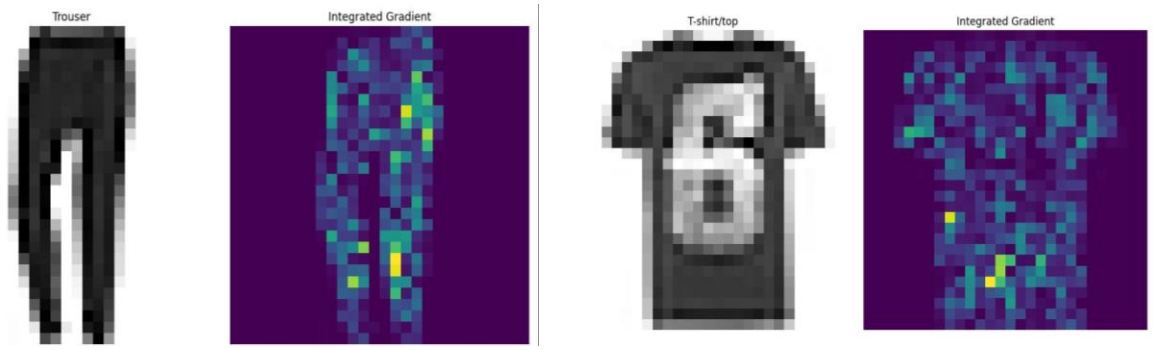

Figure 5 Few outputs of Integrated Gradients Technique on the FashionMNIST dataset

We loaded a pretrained model from Huggingface [5] and computed gradients on sample digit images from SVHN dataset. The relevant code is as follows:

```
inputFile = "input/6.png"
referenceImageFile = "input/blank.png"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
idx = 6
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = timm.create_model("resnet18_svhn", pretrained=True)
model.to(device)
model.eval()
input = Image.open(inputFile).convert("RGB")
referenceImage = Image.open(referenceImageFile).convert("RGB")
input = transform_test(input).unsqueeze(0)
referenceImage = transform_test(referenceImage).unsqueeze(0)
print("input:"+str(input.shape))
print("referenceImage:"+str(referenceImage.shape))
input = input.to(device)
referenceImage = referenceImage.to(device)
gradients = integrated_gradients(input, referenceImage, model, idx)
gradients = gradients.squeeze(0)
gradients = gradients.permute(1,2,0)
imagerecon = input.squeeze(0)
imagerecon = transforms.Normalize((0,0,0), (1/0.229, 1/0.224, 1/0.225))(imagerecon)
imagerecon = transforms.Normalize((-0.485, -0.456, -0.4060), (1, 1, 1))(imagerecon)
imagerecon = imagerecon.permute(1,2,0)
gradientsCpu = np.fabs(gradients.cpu().data.numpy())
```

```
gradientsCpu = gradientsCpu / np.max(gradientsCpu)
images = [imagerecon.cpu().data.numpy(), gradientsCpu]
save_and_show_output(images, "6", "output/test_resnet_"+ inputFile.split('/')[-1])
```

A few sample outputs of the code are shown in Figure 6. The inputs to this algorithm are manually uploaded files.

Figure 6 Few instances of SVNH images on which integrated gradients is applied.

Key Resources and Deliverables-

1 – Accumuated Gradients approach takes inspiration from the project member's previous research present in [1], while Integrated Gradients is inspired from the paper [2] and GradCAM/EigenCAM is inspired from the paper [3]

2 – The pretrained models used in the code are Resnet18 [4] and Resnet18-SVNH [5].

3 – Entire code is opensource and is viewable at - https://github.com/choprahetarth/CV-Feature-Importance. The videos can be viewed at - https://github.com/choprahetarth/CV-Feature-Importance/tree/main/Accumulated-Gradients/videos .

Plan for the Remainder of the Project -

We are interested in exploring the following points

1 – Studying the Stochasticity of the Global Feature Importances, how the variance is affected by batch size, initialization, averaging, as well as on a bigger dataset.

2 – Using GAN inspired techniques/DeepLift to explore local feature importance.

3 – Compile and clean the code for a neater software library like implementation.

Member Roles –

1 – Hetarth - Worked on Accumulated Gradients.

2 – Avinoor – Worked on EigenCAM and assisted Hetarth with Accumulated Gradients for the video exporting.

3 – Tarun – Worked on Integrated Gradients.

We are open to feedback / other ideas / applications other than classification, for the project! The code is not in a good state currently and will be made adaptable soon for everyone to import and use

[1] H. Chopra, "Global NN Feature Selection," Kaggle, [Online]. *Available: https://www.kaggle.com/code/hetarthchopra/global-nn-feature-selection. [Accessed: 7 March 2024]*.

[2] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic Attribution for Deep Networks," in Proc. *34th Int. Conf. Mach. Learn. (ICML), Sydney, NSW, Australia, 2017, vol. 70, pp. 3319-3328.*

[3] Muhammad, Mohammed Bany, and Mohammed Yeasin. "Eigen-cam: Class activation map using principal components." *In 2020 international joint conference on neural networks (IJCNN), pp. 1-7. IEEE, 2020.*

[4] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." *In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.*

[5] HuggingFace, “Resnet18_SVHN,” huggingface.co, [Online]. *Available :* https://huggingface.co/edadaltocg/resnet18_svhn/tree/main*. [Accessed : 7 March 2024]*