DATA AUGMENTATION TECHNIQUES USING CIFAR10 DATASET and FMNIST

DATA:

Cifar10 dataset:

```
plt.figure(figsize=(10, 2))
for i, idx in enumerate(random_indices, 1):
    plt.subplot(1, 5, i)
    plt.imshow(x_train[idx])
    plt.axis('off')

plt.tight_layout()
plt.show()
```

This code displays 5 randomly selected CIFAR-10 dataset images in a horizontal row, each with its index in a subplot. The images are shown without axis labels.

Data Augmentation Techniques Used:

1.Normalization

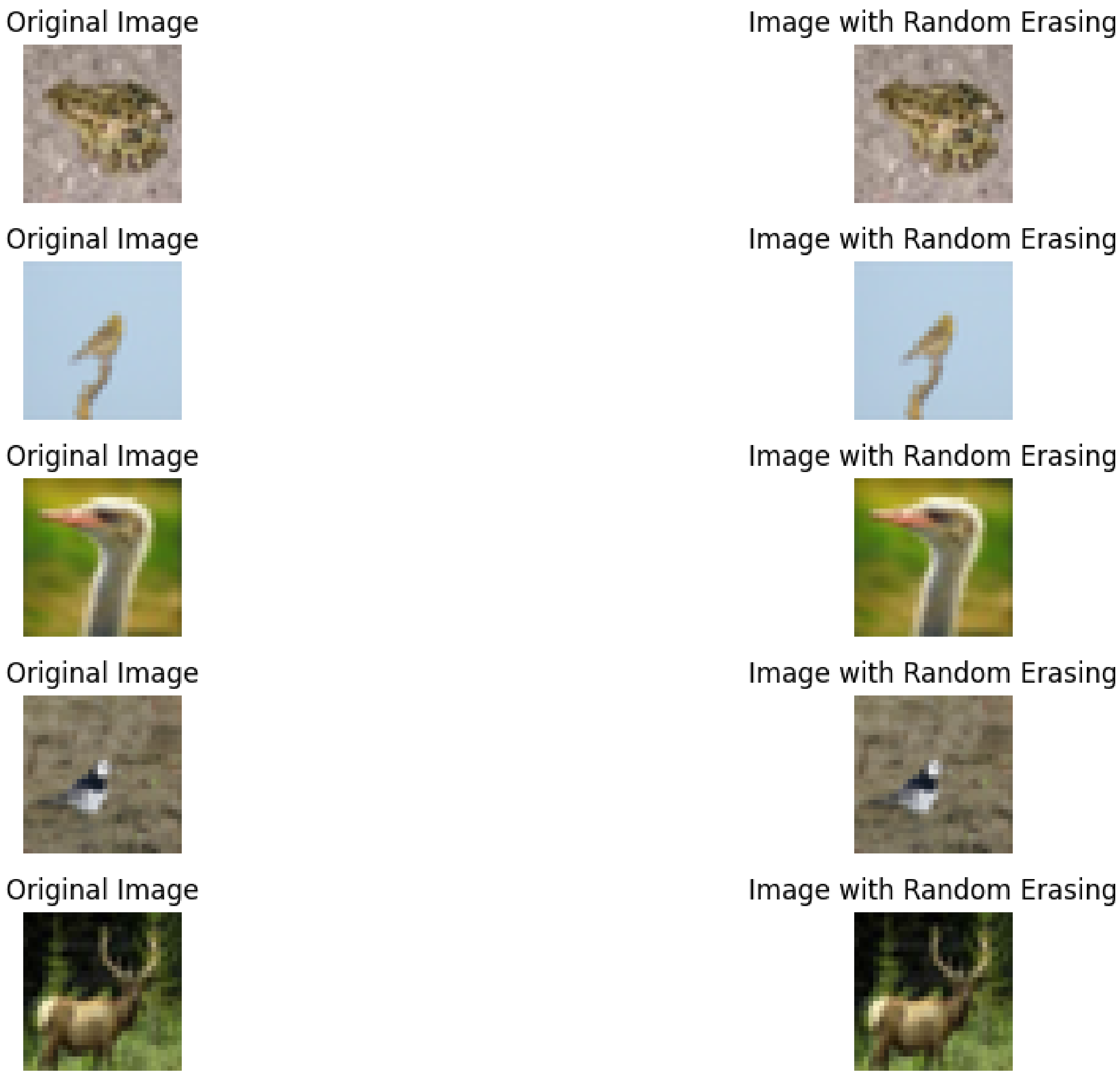
```
#Data Augmentation Function: Let's define an instance of the ImageDataGenerator class and set the parameters.We have t
train_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip=True,
                                    zoom_range=.1 )

val_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip=True,
 (variable) test_generator: ImageDataGenerator .1)

test_generator = ImageDataGenerator(
                                    rotation_range=2,
                                    horizontal_flip= True,
                                    zoom_range=.1)
```

Data augmentation is a technique used in machine learning and deep learning, particularly in computer vision tasks. It involves creating new training data by applying various transformations to existing examples. These transformations include rotations, flips, translations, scaling, cropping, and changes in brightness, contrast, or color. Data augmentation aims to increase the diversity of the training dataset, which helps improve the model's ability to generalize and perform well on new, unseen data. By presenting the model with a wider range of variations, it learns to recognize features that are invariant to these changes, leading to enhanced robustness and reduced overfitting. Data augmentation is especially valuable when the available training data is limited, as it effectively increases the effective size of the dataset and can lead to more accurate and reliable machine learning models.

2.Random Eraser



The `**random_erasing**` function performs a data augmentation technique often used in computer vision tasks. It enhances neural network training by randomly erasing a rectangular region in an input image. The probability parameter controls the likelihood of applying the transformation. A target area and aspect ratio are randomly selected within specified ranges. The function then calculates the erasing rectangle's dimensions and position, ensuring they fit within the image. This region is replaced with random pixel values, simulating occlusion. This process encourages the network to learn more robust features and reduces overfitting. By introducing controlled variations, the function aims to improve the model's generalization performance during training, ultimately enhancing its ability to handle diverse real-world scenarios.

3.Salient

```
def normalize_image(img):
    grads_norm = img[:,:,0]+ img[:,:,1]+ img[:,:,2]
    grads_norm = (grads_norm - tf.reduce_min(grads_norm))/ (tf.reduce_max(grads_norm)- tf.reduce_min(grads_norm))
    return grads_norm
def saliency(img1, img2,vmin=0.3,vmax=0.7, mix_val=2):
    '''
    input:
    ------
    img1=gradient image
    img2= original image
    output:
    -------
    img_new
    '''
    img_new = img1*mix_val+img2/mix_val
    return img_new
def grads_saliency(img, test_model):
    input_img = img
    input_img = tf.keras.applications.densenet.preprocess_input(input_img)
    result = test_model(input_img)
    max_idx = tf.argmax(result,axis = 1)
    tf.keras.applications.imagenet_utils.decode_predictions(result.numpy())
    with tf.GradientTape() as tape:
        tape.watch(input_img)
        result = test_model(input_img)
        max_score = result[0,max_idx[0]]
    grads = tape.gradient(max_score, input_img)
    return grads
```

```
def saliency_image(img):
    # img = array_to_img(img, scale=False)
    # img = img.resize((224, 224))  # Resize images to (224, 224)
    # img = img_to_array(img)
    img_grads = grads_saliency(tf.convert_to_tensor(img[None,...]), test_model)
    img_saliency = saliency(img_grads, img,vmin=0.3,vmax=0.7, mix_val=2)
    return img_saliency

train_datagen = ImageDataGenerator(
    rotation_range=2,
    horizontal_flip=True,
    zoom_range=0.1,
    preprocessing_function=saliency_image,
    validation_split=0.3
)

val_datagen = ImageDataGenerator(
    rotation_range=2,
    horizontal_flip=True,
    zoom_range=0.1
)

test_datagen = ImageDataGenerator(
    rotation_range=2,
    horizontal_flip=True,
    zoom_range=0.1
)
```

The provided code defines a series of functions for generating saliency images, which highlight regions in an input image that contribute most to the output of a neural network model. Saliency maps are a form of data augmentation used for visualizing model attention and understanding decision-making.

The `normalize_image` function takes an image as input and computes the normalized gradient magnitudes across color channels. The values are normalized to the range [0, 1].

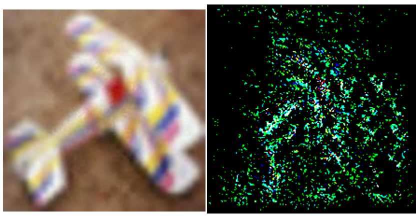
The `saliency` function blends the gradient image (img1) and the original image (img2) to create a new image (img_new). The blending factor is controlled by the parameter `mix_val`, and the output image is normalized within the range specified by `vmin` and `vmax`.

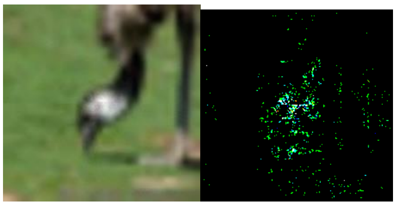
The `grads_saliency` function computes the gradient of the highest predicted score in the model's output with respect to the input image. This gradient indicates the importance of each pixel for the predicted class.

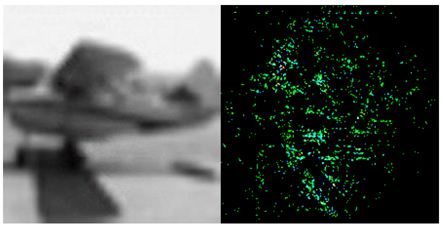
The `saliency_image` function calculates the saliency map for an input image using the `grads_saliency` function and enhances it using the `saliency` function.

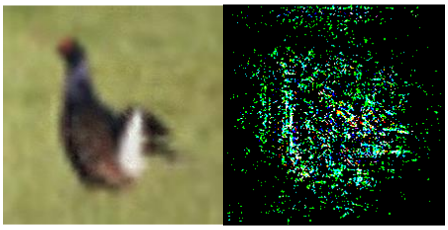
In summary, the provided functions enable the visualization of regions that significantly influence the output of a neural network, offering insights into how the model makes predictions. This type of data augmentation aids in understanding the model's decision-making process and identifying areas where its attention is focused.

The left side are normal image while the right image is salient image.









Data Preprocessing:

```
#Divide the data in Train, Validation and Test Datasets
(x_train,y_train),(x_test,y_test)=cifar10.load_data()
```

```
x_train,x_val,y_train,y_val=train_test_split(x_train,y_train,test_size=.3)
✓ 0.0s
```

Splitting training data for validation with 30% size to assess model training.

```
#Print the dimensions of the datasets to make sure everything's kosher

print((x_train.shape,y_train.shape))
print((x_val.shape,y_val.shape))
print((x_test.shape,y_test.shape))
```

✓ 0.0s

```
((35000, 32, 32, 3), (35000, 1))
((15000, 32, 32, 3), (15000, 1))
((10000, 32, 32, 3), (10000, 1))
```

Code checks and displays data dimensions for training, validation, and testing sets.

```
#One hot encode the labels.Since we have 10 classes we should expect the shape[1] of

y_train=to_categorical(y_train)
y_val=to_categorical(y_val)
y_test=to_categorical(y_test)
```

✓ 0.0s

Converts categorical labels into one-hot encoded format, changing shape[1] from 1 to 10 for each y_train, y_val, and y_test dataset. This is used for multi-class classification.

```
# Lets print the dimensions one more time to see if things changed the way we expected

print((x_train.shape,y_train.shape))
print((x_val.shape,y_val.shape))
print((x_test.shape,y_test.shape))
```

✓ 0.0s

```
((35000, 32, 32, 3), (35000, 10))
((15000, 32, 32, 3), (15000, 10))
((10000, 32, 32, 3), (10000, 10))
```

Neural Networks.

```
'The first base model used is VGG19. The pretrained weights from the imagenet challenge are used'
base_model_1 = VGG19(include_top=False,weights='imagenet',input_shape=(32,32,3),classes=y_train.shape[1])

'For the 2nd base model we will use Resnet 50 and compare the performance against the previous one.The hypothesis is that Re
base_model_2 = ResNet50(include_top=False,weights='imagenet',input_shape=(32,32,3),classes=y_train.shape[1])

'For the 3rd base model we will use DenseNet121 and compare the performance against the previous one.The hypothesis is that
base_model_3 = DenseNet121(include_top=False,weights='imagenet',input_shape=(32,32,3),classes=y_train.shape[1])

'For the 4th base model we will use MobileNetV2 and compare the performance against the previous one.'
base_model_4 = MobileNetV2(include_top=False,weights='imagenet',input_shape=(32,32,3),classes=y_train.shape[1])

'For the 5th base model we will use EfficientNetB0 and compare the performance against the previous one.'
base_model_5 = EfficientNetB0(include_top=False,weights='imagenet',input_shape=(32,32,3),classes=y_train.shape[1])
```

This code snippets defines all the model used in here.

```
lrr= ReduceLROnPlateau(
                       monitor='val_acc', #Metric to be measured
                       factor=.01, #Factor by which learning rate
                       patience=3,  #No. of epochs after which if
                       min_lr=1e-5) #The minimum learning rate
```

```
batch_size= 100
epochs=50
```

This code Snippet defines the **batch_size** and **epochs.**

The code snippets below show the Layers used in the different base_models defined before.

1.VGG19

```
#Lets add the final layers to these base models where the actual classification is done in the dense layers

model_1= Sequential()
model_1.add(base_model_1) #Adds the base model (in this case vgg19 to model_1)
model_1.add(Flatten()) #Since the output before the flatten layer is a matrix we have to use this function to get a vec
```

```
#Add the Dense layers along with activation and batch normalization
model_1.add(Dense(1024,activation=('relu'),input_dim=512))
model_1.add(Dense(512,activation=('relu')))
model_1.add(Dense(256,activation=('relu')))
#model_1.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_1.add(Dense(128,activation=('relu')))
#model_1.add(Dropout(.2))
model_1.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

2.ResNet50

```
#Since we have already defined Resnet50 as base_model_2, let us build the sequential model.

model_2=Sequential()
#Add the Dense layers along with activation and batch normalization
model_2.add(base_model_2)
model_2.add(Flatten())


#Add the Dense layers along with activation and batch normalization
model_2.add(Dense(4000,activation=('relu'),input_dim=512))
model_2.add(Dense(2000,activation=('relu')))
model_2.add(Dropout(.4))
model_2.add(Dense(1000,activation=('relu')))
model_2.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_2.add(Dense(500,activation=('relu')))
model_2.add(Dropout(.2))
model_2.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

3.DenseNet121

```
#Since we have already defined DenseNet121 as base_model_3, let us build the sequential model.

model_3=Sequential()
#Add the Dense layers along with activation and batch normalization
model_3.add(base_model_3)
model_3.add(Flatten())


#Add the Dense layers along with activation and batch normalization
model_3.add(Dense(4000,activation=('relu'),input_dim=512))
model_3.add(Dense(2000,activation=('relu')))
model_3.add(Dropout(.4))
model_3.add(Dense(1000,activation=('relu')))
model_3.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_3.add(Dense(500,activation=('relu')))
model_3.add(Dropout(.2))
model_3.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

4.MobileNetV2

```
#Since we have already defined DenseNet121 as base_model_3, let us build the sequential model.

model_4=Sequential()
#Add the Dense layers along with activation and batch normalization
model_4.add(base_model_4)
model_4.add(Flatten())


#Add the Dense layers along with activation and batch normalization
model_4.add(Dense(4000,activation=('relu'),input_dim=512))
model_4.add(Dense(2000,activation=('relu')))
model_4.add(Dropout(.4))
model_4.add(Dense(1000,activation=('relu')))
model_4.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_4.add(Dense(500,activation=('relu')))
model_4.add(Dropout(.2))
model_4.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

5.EfficientNet

```
#Since we have already defined EfficientNet as base_model_5, let us build the sequential model.

model_5=Sequential()
#Add the Dense layers along with activation and batch normalization
model_5.add(base_model_5)
model_5.add(Flatten())


#Add the Dense layers along with activation and batch normalization
model_5.add(Dense(4000,activation=('relu'),input_dim=512))
model_5.add(Dense(2000,activation=('relu')))
model_5.add(Dropout(.4))
model_5.add(Dense(1000,activation=('relu')))
model_5.add(Dropout(.3))#Adding a dropout layer that will randomly drop 30% of the weights
model_5.add(Dense(500,activation=('relu')))
model_5.add(Dropout(.2))
model_5.add(Dense(10,activation=('softmax'))) #This is the classification layer
```

Results:

Case1: Normal Data Augmentation

1.Accuracy Snippet

1.VGG19

```
Epoch 47/50
350/350 [==============================] - ETA: 0s - loss: 0.0234 - accuracy: 0.9919WARNING:tensorflow:Lear
350/350 [==============================] - 25s 70ms/step - loss: 0.0234 - accuracy: 0.9919 - lr: 0.0010
```

2.ResNet50

```
Epoch 100/100
350/350 [==============================] - ETA: 0s - loss: 0.3339 - accuracy: 0.8898WARNING:tensorflow:Lear
350/350 [==============================] - 27s 77ms/step - loss: 0.3339 - accuracy: 0.8898 - lr: 0.0010
```

3.DenseNet121

```
Epoch 100/100
350/350 [==============================] - ETA: 0s - loss: 0.7307 - accuracy: 0.7546WARNING:tensorflow:Learning r
350/350 [==============================] - 31s 87ms/step - loss: 0.7307 - accuracy: 0.7546 - lr: 0.0010
```

4.MobileNetV2

```
Epoch 99/100
350/350 [==============================] - ETA: 0s - loss: 0.6663 - accuracy: 0.7801WARNING:tensorflow:Learning rate reducti
350/350 [==============================] - 24s 70ms/step - loss: 0.6663 - accuracy: 0.7801 - lr: 0.0010
```

5.EfficientNetB0

```
Epoch 41/100
350/350 [==============================] - ETA: 0s - loss: 0.5640 - accuracy: 0.8171WARNING:tensorflow:Learning ra
350/350 [==============================] - 26s 74ms/step - loss: 0.5640 - accuracy: 0.8171 - lr: 0.0010
Epoch 42/100
```

Case2: Random Images

1.Accuracy Snippet

1.VGG19

```
Epoch 49/50
350/350 [==============================] - ETA: 0s - loss: 0.0235 - accuracy: 0.9922WARNING:tensorflow:Learning
350/350 [==============================] - 25s 70ms/step - loss: 0.0235 - accuracy: 0.9922 - lr: 0.0010
```

2.ResNet50

```
Epoch 49/50
350/350 [==============================] - ETA: 0s - loss: 0.1271 - accuracy: 0.9616WARNING:tensorflow:Lear
350/350 [==============================] - 26s 74ms/step - loss: 0.1271 - accuracy: 0.9616 - lr: 0.0010
```

3.DenseNet121

```
Epoch 47/50
350/350 [==============================] - ETA: 0s - loss: 2.3027 - accuracy: 0.1003WARNING:tensorflow:Lea
350/350 [==============================] - 31s 88ms/step - loss: 2.3027 - accuracy: 0.1003 - lr: 0.0010
```

4.MobileNetV2

```
Epoch 46/50
350/350 [==============================] - ETA: 0s - loss: 0.7846 - accuracy: 0.7393WARNING:tensorflow:Lea
350/350 [==============================] - 24s 69ms/step - loss: 0.7846 - accuracy: 0.7393 - lr: 0.0010
```

5.EfficientNetB0

```
Epoch 50/50
350/350 [==============================] - ETA: 0s - loss: 0.5243 - accuracy: 0.8305WARNING:tensorflow:L
350/350 [==============================] - 27s 76ms/step - loss: 0.5243 - accuracy: 0.8305 - lr: 0.0010
```

Case3: Salience

1.Accuracy Snippet

1.VGG19

```
Epoch 17/50
20/20 [==========================] - ETA: 0s - loss: 0.0232 - accuracy: 0.9920 WARNING:tensorflow:Learning rate reduction is conditioned on me
20/20 [==========================] - 1102s 57s/step - loss: 0.0232 - accuracy: 0.9920 - val_loss: 1.0603 - val_accuracy: 0.7800 - lr: 0.0010
```

2.ResNet50

```
Epoch 46/50
350/350 [==============================] - ETA: 0s - loss: 0.2181 - accuracy: 0.9356WARNING:tensorflow:Le
350/350 [==============================] - 26s 75ms/step - loss: 0.2181 - accuracy: 0.9356 - lr: 0.0010
```

3.DenseNet121

```
Epoch 48/50
350/350 [==============================] - ETA: 0s - loss: 0.0776 - accuracy: 0.9734WARNING:tensorflow:Le
350/350 [==============================] - 25s 71ms/step - loss: 0.0776 - accuracy: 0.9734 - lr: 0.0010
```

4.MobileNetV2

```
Epoch 48/50
350/350 [==============================] - ETA: 0s - loss: 0.3416 - accuracy: 0.8917WARNING:tensorflow:Lea
350/350 [==============================] - 26s 74ms/step - loss: 0.3416 - accuracy: 0.8917 - lr: 0.0010
```

5.EfficientNetB0

```
Epoch 47/50
350/350 [==============================] - ETA: 0s - loss: 0.2274 - accuracy: 0.9347WARNING:tensorflow:Lear
350/350 [==============================] - 26s 74ms/step - loss: 0.2274 - accuracy: 0.9347 - lr: 0.0010
```