Neural Network Model Powered Animal Image Synthesis Engine

Yihao Shu shu.yih@northeastern.edu
(Bruce) Xianwei Long long.xianw@northeastern.edu

Note:This report only displays some key blocks of code. For the complete code, please refer to the provided code file.

BACKGROUND

The Problem

Neural Networks are widely used for image recognition and tasks involved in processing pixel data. Compared to Fully-connected Neural Networks, CNN is better at capturing the features of images more effectively. By making use of this, it is possible to generate images by giving certain prompts.

A designer needs to deal with a lot of image work every day. They need to create images with some specific ideas or modify them to combine the features from other given images. In the traditional way, it is hard for designers to create or modify an image. It is highly abstractive and requires rich creativity with powerful tools - either hand sketch skill or digital drawing kits. As a result, these creative works usually move at a low pace. For example, it takes time for designers to generate ideas for some creatures that never exists, or to combine features from different animals.(figure 1)

Figure 1. Designing Creatures that Never Exist. Or Combine Features of Cat and Eagle

Therefore, it would be highly productive if there is a tool that can generate images when users input the name of animals that they want to capture the features from. For instance, by inputting

the prompt of ‘cat’ and ‘eagle’, the tool can generate an image of a creature that combines both cat and eagle’s features, let’s say a creature with cat’s body and eagle’s head.

In this project, by utilizing machine learning techniques, we are trying to develop a model that can help generate images with a given animal dataset. This is a preliminary attempt at a creative production tool.

A Potential Solution

In this project, we plan to train a Convolutional Neural Network(CNN) model and fine tune it with Out-of-Domain via Augment Targets. The CNN will be designed and then trained by large amounts of animal images with labels. The labeled images come from the iWildCam dataset(will be introduced in the next session). With the trained CNN model, we can figure out the best augmented data. The augmented features will be applied to further train the Generative adversarial networks(GAN). When the model is well-established, we can use it to generate images - in this project - to produce images with different features according to labels(The label refers to the name of the animal species). Below is a brief flow diagram of this project(Figure 2):

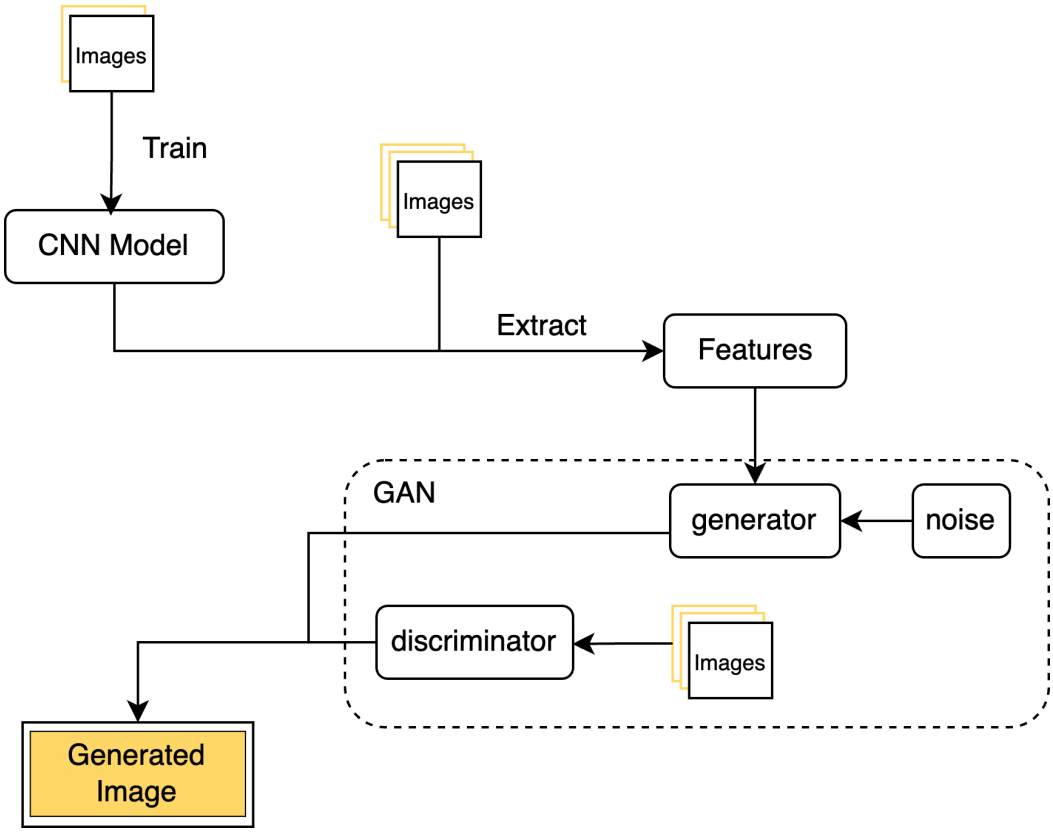


Figure 2. The Simplified Project Flow Diagram

APPROACH and IMPLEMENTATION

The Preparation of Dataset

iWildCam is the dataset which contains the animal images. Images are collected from camera traps - heat or motion-activated static cameras which are deployed by ecologists and placed in the wild to monitor wildlife biodiversity loss.

The dataset can be loaded to Python environment with below code:

```
!pip install wilds
```

```
from wilds import get_dataset
from wilds.common.data_loaders import get_train_loader
import torchvision.transforms as transforms

# Load the full dataset, and download it if necessary
dataset = get_dataset(dataset="iwildcam", download=True)
```

An example of the images are provided below in Figure 3. Dataset has been splitted into training set, ID testset and OOD test set.

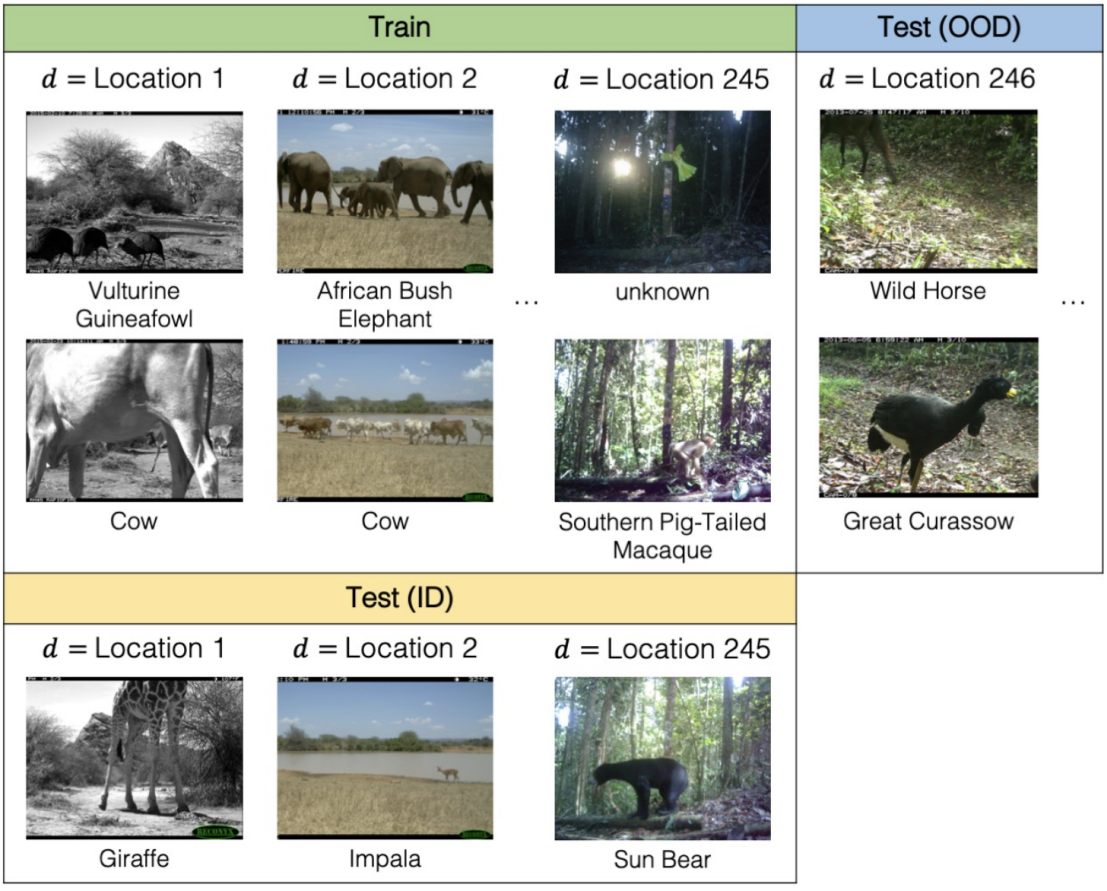


Figure 3. Images examples in iWildCam dataset.

The iWildCam dataset contains images taken from 245 cameras(location). Every image is marked in one of the 182 labels(species of animal or 'empty'). There are 12GB labeled training images in total. Each image is in torch.Size([448, 448]).

Design the CNN Model

To capture the feature of images, we design Convolutional Neural Network model as below:

```
class CNN(nn.Module):
    def __init__(self):
        super().__init__()

        self.conv1 = nn.Conv2d(3, 64, 3, 1, 1)
        self.conv2 = nn.Conv2d(64, 128, 3, 1, 1)
        self.conv3 = nn.Conv2d(128, 256, 3, 1, 1)

        self.bn1 = nn.BatchNorm2d(64)
        self.bn2 = nn.BatchNorm2d(128)
        self.bn3 = nn.BatchNorm2d(256)

        self.pool = nn.MaxPool2d(2, 2)

        self.fc1 = nn.Linear(256 * 56 * 56, 1024)
        self.fc2 = nn.Linear(1024, 182)

    def forward(self, x):

        x = self.pool(F.relu(self.bn1(self.conv1(x))))
        x = self.pool(F.relu(self.bn2(self.conv2(x))))
        x = self.pool(F.relu(self.bn3(self.conv3(x))))

        x = x.view(x.size(0), -1)  # Flatten

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Here we use the basic structure as other CNN models, with 3 convolutional layers, 3 batch normalization layers for 64, 128, 256 feature maps separately, and 2 fully-connected layers as well as max pooling layer used multiple times. The application of batch normalization layers can make the training process faster and more stable through normalization of the inputs by recentering and rescaling.

Train and Test the Model

Due to the size of the dataset, it takes 2.5 hours time and 3GB storage to train a model for one camera(location). With limited computational resources, we only train models for the first 10 cameras(location). For each camera, we train the model with 5 epochs. We use cross-entropy loss and Adam optimizer to evaluate and optimize the model while training.

The original test data set is using Location246 in iWildCam, which is not our target. We aim to

test the model accuracy for each location. Therefore, we split the dataset into 80% train and 20% test in each location.

Below is the result of for location 0 to 9:

```
Location: 0, Accuracy: 1.0000, Loss: 0.0000
Location: 9, Accuracy: 1.0000, Loss: 0.0000
Location: 8, Accuracy: 0.9483, Loss: 0.7958
Location: 1, Accuracy: 0.8023, Loss: 0.5085
Location: 6, Accuracy: 0.7273, Loss: 1.5323
Location: 2, Accuracy: 0.6052, Loss: 1.7558
Location: 5, Accuracy: 0.5415, Loss: 1.2295
Location: 3, Accuracy: 0.0000, Loss: 0.0000
Location: 4, Accuracy: 0.0000, Loss: 0.0000
Location: 7, Accuracy: 0.0000, Loss: 0.0000
```

We plan to choose location 5 to further train the GAN model. However, due to the low accuracy in location 5 model, it is necessary to improve the performance via Out of Domain Method.

Out of Domain Method

In the last session, we chose location 5 to further train a GAN model. The CNN model trained by the images in location 5 is of low accuracy, its performance needs to be improved mainly through the augmentation for the image data:

> Use Transforms in Torch to randomly rotate the image, flip the image and change the features such as brightness, contrast and saturation. The mean and std is set as the same as the images in the data loader.

```python
from torchvision import transforms

# Define a robust augmentation pipeline
augmentation = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

> Below is same samples of images after augmentation:

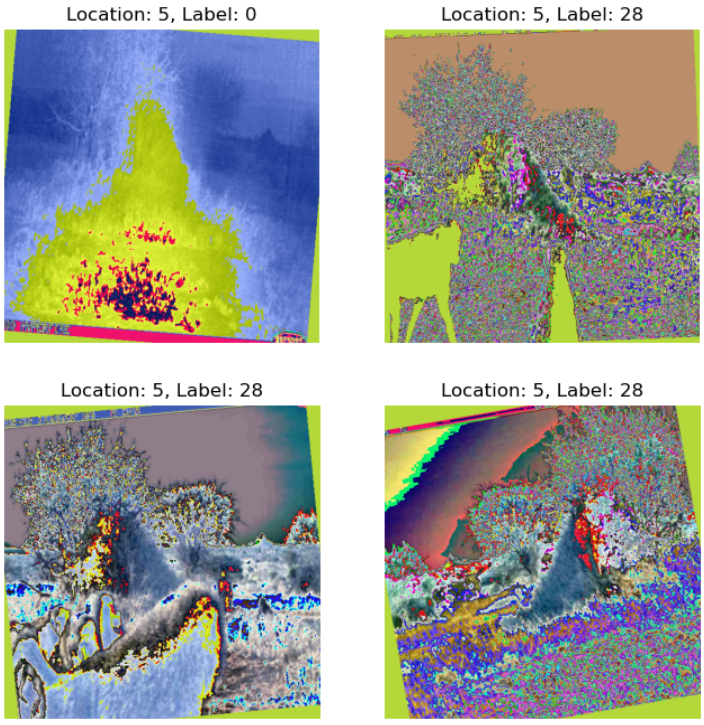


Figure 4. Samples of images after augmentation

With the augmented images, the adversarial method is applied in the training. Here we use FGSM (PGD can be used instead).

```
def fgsm(image, alpha, data_grad):
    perturbed_image = image + alpha * data_grad.sign()
    perturbed_image = torch.clamp(perturbed_image, min=0, max=1)
    return perturbed_image
```

To prevent overfitting during training, we introduce L2 regularization in the Adam Optimizer through weight decay.

```
# L2 regularization
optimizer = torch.optim.Adam(model.parameters(), lr=0.001, weight_decay=1e-5)
```

After training, we find out that the test accuracy on the test dataset is at a low level of 26%. Considering the low accuracy, we drew the conclusion:

- With FGSM and alpha = 0.1, it causes a lot of noise in the images, leading to overfitting to the noises.
- Due to the weakness of our current CNN model, it is hard to capture all the features in such a complex dataset.
- Based on the fail experience, we plan to implement the second fine tuning OOD method,

by setting the alpha in adversarial training lower and choosing an enhanced CNN model.

Below is our second attempt to train for OOD.

Again, we augment the images data. Compared to the previous augmentation, this time we just keep the random flip and rotate. The mean and std from the paper Out-of-Domain Robustness via Targeted Augmentations[1] are used.

```
# Define a new augmentation pipeline
augmentation = transforms.Compose([
    transforms.ToPILImage(),
    transforms.RandomHorizontalFlip(),
    #transforms.ColorJitter(brightness=0.5),
    transforms.RandomRotation(15),
    transforms.ToTensor(),
    transforms.Normalize(mean=mean, std=std)
])
```

The new augmented images are shown below:

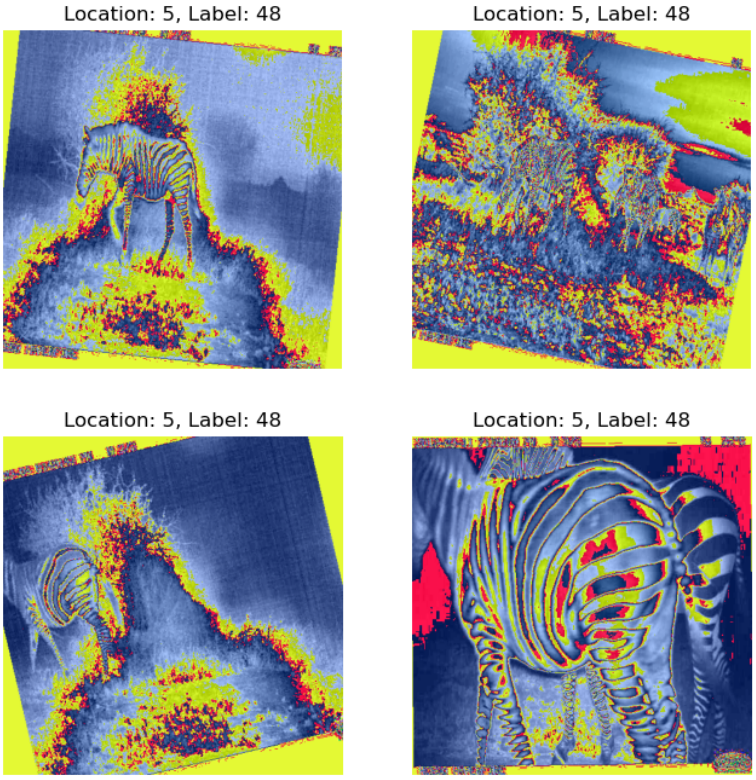


Figure 5. Some samples of augmented images

The CNN model is also modified. One more convolutional layer and batch normalization layer is added to enhance the ability of capturing features. A dropout layer is added to avoid overfitting. Number of epochs is increased to better train the model.

After training the new-designed model with the augmented images, the accuracy of the model has a 5% improvement.

GAN Model Design

With the CNN model in the last step, we figure out the best augmented image data. The augmented features extracted from the images are used to train the GAN model.

In the GAN model, it consists of two neural network models: a generator and a discriminator. The generator is used to create images that are indistinguishable from real images while the discriminator aims to differentiate the real and fake images. In the GAN model, we use LeakyReLU instead of ReLU to ensure gradients do not become sparse during training. The Tanh is used to ensure the pixel values of the generated images as real ones.
These two components are defined below:

```
class ConditionalGenerator(nn.Module):
    def __init__(self, noise_dim, feature_dim, output_dim):
        super(ConditionalGenerator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(noise_dim + feature_dim, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm1d(256),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm1d(512),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.BatchNorm1d(1024),
            nn.Linear(1024, output_dim),
            nn.Tanh()
        )

    def forward(self, noise, features):
        x = torch.cat([noise, features], dim=1)
        return self.model(x)
```

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(3 * 448 * 448, 1024),  # Adjusted input dimension
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity
```

GAN Model Training

Below is the main component about how we train the GAN model:

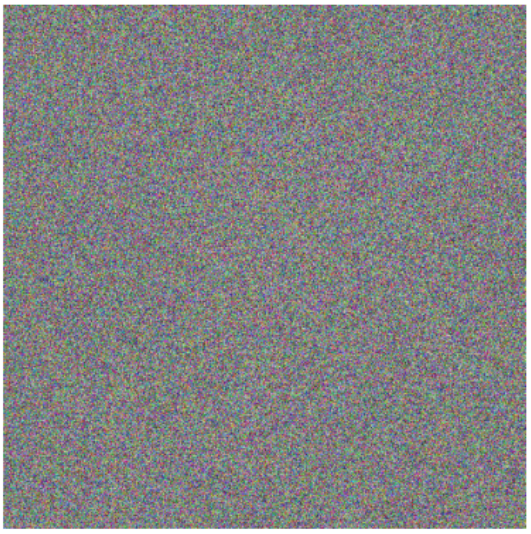
- real_labels: A tensor of ones, signifying that the images are real.
- fake_labels: A tensor of zeros, indicating that the images are generated (fake).
- The discriminator is trained twice in each loop: once with real images and once with fake images generated by the generator.
- The generator's goal is to fool the discriminator. It generates images using the same random noise and features.
- Adversarial Loss: A binary cross-entropy, the discriminator is trained to minimize the loss while the generator is trained to maximize the loss when the discriminator correctly identifies fake data.

The training loop is designed to achieve a balance where the generator produces realistic images, and the discriminator is adept at distinguishing real from fake. When the epochs increase, the performance of the model improves.

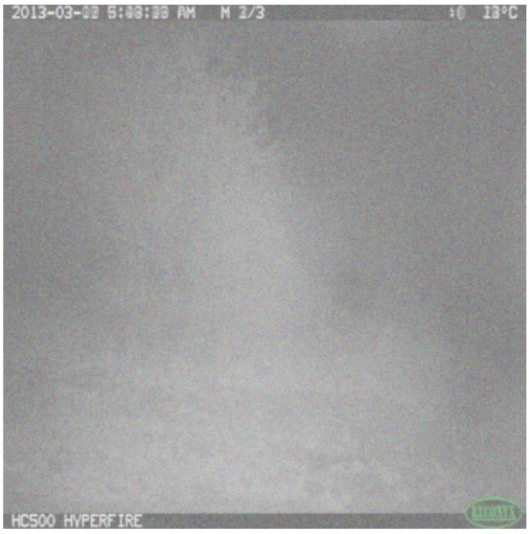
Results

Note that we use lable48(zebra) to generate images with the model.

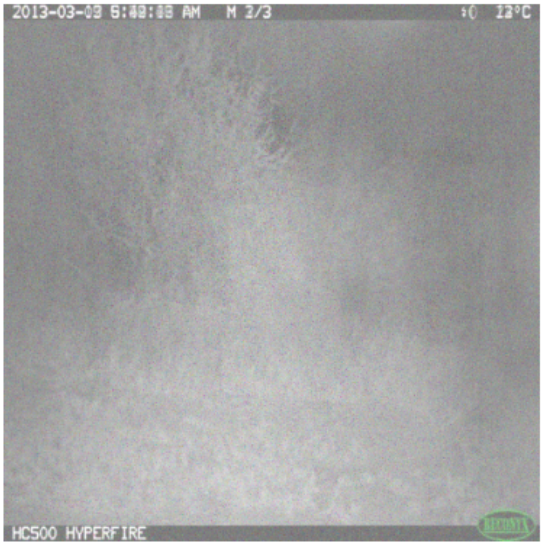
1 epoch:



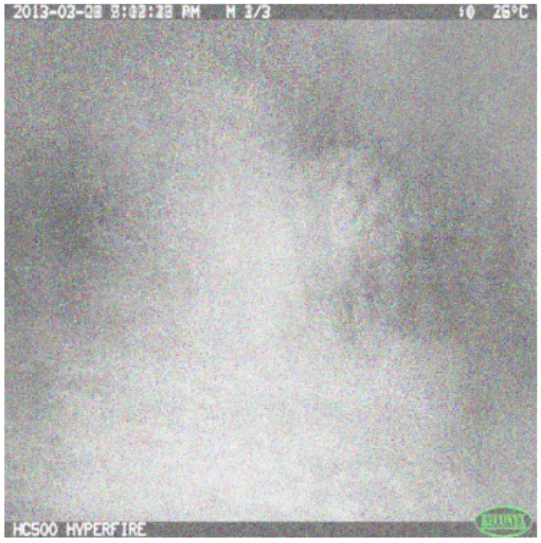
50 epoches:



175 epoches:



225 epoches:



325 epoches:

445 epoches:

From the image, we discover that as time of training increases, the more vivid the image can be by the model. Note that there is still an ambiguity in data and time, which is due to every label's image having a different complexity. In epoch 445, we finally get an image of a zebra.

DISCUSSION

Conclusion

- Parameter choosing is the key to success in Neural Network model training
- Due to the imperfection in choosing the proper parameters, the image generated by the model is not ideal
- More epochs in training usually lead to a better generator model performance. It also requires more time and computational resources

Future Work

- Figure out better parameters to train the model
- Apply GANcycle

Reference

[1] Gao, Irena, et al. "Out-of-Domain Robustness via Targeted Augmentations." *arXiv preprint arXiv:2302.11861* (2023) *[Online]. https://arxiv.org/abs/2302.11861.*