CONTROL NET

```
import torch
import torchvision.transforms as transforms
import torchvision.models as models
import numpy as np
from PIL import Image
import os
import matplotlib.pyplot as plt

# Function to load image from a folder
def load_images_from_folder(folder_path):
    images = []
    for filename in os.listdir(folder_path):
        image_path = os.path.join(folder_path, filename)
        if os.path.isfile(image_path):
            image = Image.open(image_path).convert('RGB')
            images.append((image, filename))
    return images

# Define transformation to tensor
transform = transforms.Compose([
    transforms.RandomHorizontalFlip(p=0.5),
    transforms.RandomVerticalFlip(p=0.5),
    transforms.RandomRotation(degrees=45),
    transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.5),
    transforms.RandomResizedCrop(size=(224, 224), scale=(0.8, 1.0)),  # Normalization for ResNet
    transforms.ToTensor()
])

# Define and train the control net
class ControlNet(torch.nn.Module):
    def _init_(self):
        super(ControlNet, self)._init_()
        self.resnet = models.resnet18(pretrained=True)
        self.resnet.fc = torch.nn.Linear(self.resnet.fc.in_features, 1)

    def forward(self, x):
        return torch.sigmoid(self.resnet(x)).squeeze()

# Apply control net
def apply_control_net(image_tensor, control_net):
    control_signal = control_net(image_tensor)
    augmented_image = (image_tensor * control_signal * 255).clamp(0, 255).byte()
    return augmented_image

# Load the images from input folder
input_folder = r"C:\Users\MY\Desktop\drought"
images = load_images_from_folder(input_folder)

# Initialize control net
control_net = ControlNet()

# Process each image
```

```
output_folder = r"C:\Users\MY\Desktop\aug"
os.makedirs(output_folder, exist_ok=True)

for image, filename in images:
    # Apply transformations and control net to each image
    for i in range(5):  # Apply augmentation 5 times
        image_tensor = transform(image).unsqueeze(0)
        augmented_image_tensor = apply_control_net(image_tensor, control_net)
        augmented_image = augmented_image_tensor.squeeze(0).permute(1, 2, 0).numpy().astype(np.uint8)
        output_image_path = os.path.join(output_folder, f'{os.path.splitext(filename)[0]}_{i}.jpg')
        augmented_image_pil = Image.fromarray(augmented_image)
        augmented_image_pil.save(output_image_path)

print("Augmentation process is complete. Augmented images saved in:", output_folder)
```

This Python script performs data augmentation on a set of images using a control network. Let's break down the code and explain each part:

1. **Importing Libraries:**
   - The script begins by importing necessary libraries such as `torch` for deep learning, `torchvision` for computer vision tasks, `numpy` for numerical computations, `PIL` for image processing, `os` for file operations, and `matplotlib.pyplot` for visualization.

2. **Loading Images:**
   - The `load_images_from_folder` function loads images from a specified folder. It iterates through each file in the folder, opens it as an RGB image using PIL (`Image.open`), and appends it to a list along with its filename.

3. **Image Transformation:**
   - The `transform` variable defines a series of image transformations using `torchvision.transforms.Compose`. These transformations include random horizontal and vertical flips, random rotation, color jitter, and random resized crop. The transformations are then applied to convert the image into a tensor.

4. **Defining Control Network:**
   - The `ControlNet` class defines a control network using PyTorch. It inherits from `torch.nn.Module`. The network architecture is based on a pre-trained ResNet-18 model (`models.resnet18(pretrained=True)`). The fully connected layer (`fc`) of the ResNet model is replaced with a single output neuron using `torch.nn.Linear`. The output is passed through a sigmoid activation function and squeezed to a single value.

5. **Applying Control Network:**
   - The `apply_control_net` function takes an input image tensor and the control network. It passes the image through the control network to obtain a control signal. This control signal is then used to augment the image. The pixel values of the image tensor are multiplied by the control signal, clamped between 0 and 255, and converted to bytes.

6. **Processing Images:**
   - Images are loaded from the specified input folder using `load_images_from_folder`.
   - The control network is initialized.
   - Each image is processed in a loop. For each image, the defined transformations and the control network are applied five times (`for i in range(5)`). The augmented images are saved to the output folder with filenames indicating the original image's name and the augmentation iteration.

7. **Saving Augmented Images:**
   - The augmented images are saved in the specified output folder using `Image.save`.

8. **Print Statement:**
   - Finally, a print statement confirms the completion of the augmentation process and indicates the folder where the augmented images are saved.

Overall, this script demonstrates how to use a control network to perform data augmentation on a set of images, combining traditional image transformations with learned control signals to enhance the variability and quality of the augmented dataset.

USE OF RESNET-18

In the provided code, the ResNet-18 architecture is used as the base architecture for the control network. Here's why ResNet-18 is utilized in this context:

1. **Pre-trained Model**: ResNet-18 is a convolutional neural network architecture that has been pre-trained on a large dataset (typically ImageNet). By using a pre-trained model, we can leverage the features learned by the network from a diverse set of images. This pre-training helps in capturing generalizable features, which can be beneficial for various computer vision tasks, including the control task in this case.
2. **Feature Extraction**: The ResNet architecture is designed to extract hierarchical features from images. The network consists of multiple layers, each containing convolutional blocks. These blocks help in capturing features of varying complexities, starting from simple edges and textures to more abstract and high-level concepts such as object parts and structures. By utilizing ResNet-18, we can leverage these learned features to make decisions about how to augment input images effectively.
3. **Transfer Learning**: By using a pre-trained ResNet-18 model, we benefit from transfer learning. Transfer learning allows us to transfer the knowledge gained from training on one task (e.g., image classification on ImageNet) to a different but related task (e.g., control signal prediction for image augmentation). This approach is often beneficial when we have limited labeled data for the target task, as is common in many real-world scenarios.
4. **Model Adaptation**: In the `ControlNet` class, the last fully connected layer of the ResNet-18 model (`self.resnet.fc`) is replaced with a new linear layer with a single output neuron. This adaptation allows the network to be fine-tuned specifically for the task of predicting a control signal for image augmentation. By adjusting the model's output layer, we can tailor the network to the requirements of the control task while still benefiting from the feature extraction capabilities of ResNet-18.

In summary, the ResNet-18 architecture is used in this code for its feature extraction capabilities, transfer learning benefits, and adaptability to the control task of predicting augmentation signals for input images.