## Python Files

### File: default_config.py

```
image_size = 224
batch_size = 64
num_workers = 8
pin_memory = True
normalize_mean = [0.485, 0.456, 0.406]
normalize_std = [0.229, 0.224, 0.225]
data_dir = "data"
train_dir = f"{data_dir}/train"
val_dir = f"{data_dir}/validation"
test_dir = f"{data_dir}/test"
label_map = {0: "cat", 1: "dog"}

default_transform_config = {
   "hflip": True,
   "brightness_jitter": 0.2,
   "contrast_jitter": 0.2,
   "saturation_jitter": 0.2,
   "hue_jitter": 0.0,
   "rotation": 25,  # rotation is in degrees
   "crop_scale": 0.3,  # crop_scale is like zooming in. 0.3 means zoom in by 30%
   "translate": 0.1,  # translate is like shifting the image. 0.1 means shift by 10%
   "shear": 10,  # shear is like skewing the image. 10 means shear by 10 degrees
}

no_transform_config = {
   "hflip": False,
   "brightness_jitter": 0,
   "contrast_jitter": 0,
   "saturation_jitter": 0,
   "hue_jitter": 0,
   "rotation": 0,
   "crop_scale": 1,
   "translate": 0,
   "shear": 0,
}

default_train_config = {
   "optimizer_type": "Adam",
   "learning_rate": 0.001,
   "weight_decay": 0,
   "momentum": None,
   "reg_type": "None",
   "reg_lambda": 0.001,
   "step_size": None,
   "gamma": None,
}

default_net_config = {
   "in_channels": 3,
   "num_classes": 2,
   "cv_layers": [
       {
           "out_channels": 16,
           "kernel_size": 3,
           "stride": 1,
           "padding": 1,
           "batch_norm": False,
           "max_pool": 0,
           "max_pool_stride": 1,
       },
       {
           "out_channels": 32,
           "kernel_size": 3,
           "stride": 1,
           "padding": 1,
           "batch_norm": False,
           "max_pool": 0,
           "max_pool_stride": 1,
       },
   ],
   "fc_layers": [{"out_features": 64, "batch_norm": False, "dropout_rate": 0}],
}
```

```python
default_config = {
    "label": "Default Experiment",
    "n_epochs": 120,
    "transform_config": default_transform_config,
    "train_config": default_train_config,
    "net_config": default_net_config,
}
```

---

**File: loaders.py**

```python
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from default_config import (
    image_size,
    batch_size,
    num_workers,
    pin_memory,
    normalize_mean,
    normalize_std,
    default_transform_config,
    train_dir,
)


def get_train_loader(transform_config=default_transform_config):
    hflip = transform_config["hflip"]

    brightness_jitter = transform_config["brightness_jitter"]
    contrast_jitter = transform_config["contrast_jitter"]
    saturation_jitter = transform_config["saturation_jitter"]
    hue_jitter = transform_config["hue_jitter"]

    rotation = transform_config["rotation"]
    crop_scale = transform_config["crop_scale"]
    translate = transform_config["translate"]
    shear = transform_config["shear"]

    transform_list = [transforms.Resize((image_size, image_size))]  # always resizing

    # Add transformations
    if hflip:
        transform_list.append(transforms.RandomHorizontalFlip(p=0.5))

    color_jitter_params = {}
    if brightness_jitter > 0:
        color_jitter_params["brightness"] = brightness_jitter
    if contrast_jitter > 0:
        color_jitter_params["contrast"] = contrast_jitter
    if saturation_jitter > 0:
        color_jitter_params["saturation"] = saturation_jitter
    if hue_jitter > 0:
        color_jitter_params["hue"] = hue_jitter
    if brightness_jitter > 0 or contrast_jitter > 0 or saturation_jitter > 0 or hue_jitter > 0:
        transform_list.append(transforms.ColorJitter(**color_jitter_params))

    affine_params = {}
    if rotation > 0:
        affine_params["degrees"] = rotation
    if crop_scale < 1 and rotation > 0:
        affine_params["scale"] = (1 - crop_scale, 1 + crop_scale)
    if translate > 0 and rotation > 0:
        affine_params["translate"] = (translate, translate)
    if shear > 0 and rotation > 0:
        affine_params["shear"] = (-shear, shear)
    if rotation > 0:  # rotation must be set for RandomAffine to work
        transform_list.append(transforms.RandomAffine(**affine_params))

    # Convert to tensor
    transform_list.append(transforms.ToTensor())

    # Normalize
    transform_list.append(transforms.Normalize(mean=normalize_mean, std=normalize_std))

    # Compose all transformations into a single pipeline
    train_transform = transforms.Compose(transform_list)
```

```
    # Load data
    train_data = datasets.ImageFolder(train_dir, transform=train_transform)

    # Create data loader
    train_loader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=True,
        num_workers=num_workers,
        pin_memory=pin_memory,
    )
    return train_loader


def get_test_transform():
    return transforms.Compose(
        [
            transforms.Resize((image_size, image_size)),
            transforms.ToTensor(),
            transforms.Normalize(mean=normalize_mean, std=normalize_std),
        ]
    )


def get_test_loader(dir):
    test_transform = get_test_transform()

    test_data = datasets.ImageFolder(dir, transform=test_transform)

    test_loader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=pin_memory,
    )

    return test_loader
```

---

**File: convolutionalNetwork.py**

```
from default_config import default_net_config, image_size
import torch
import torch.nn as nn


class ConvolutionalNetwork(nn.Module):
    def __init__(self, net_config=default_net_config):
        super(ConvolutionalNetwork, self).__init__()

        in_channels = net_config["in_channels"]
        num_classes = net_config["num_classes"]

        layers = []
        self.relu = nn.ReLU()

        tmp_channels = in_channels
        for config in net_config["cv_layers"]:
            out_channels = config["out_channels"]
            kernel_size = config.get("kernel_size", 3)
            stride = config.get("stride", 1)
            padding = config.get("padding", 0)
            batch_norm = config.get("batch_norm", False)
            max_pool = config.get("max_pool", 0)

            # Convolutional layer
            # - Output channels are the number of filters in the convolutional layer
            # - Kernel size is the size of the filter
            # - Stride is the step size for the filter
            # - Padding is the number of pixels to add around the input image
            layers.append(
                nn.Conv2d(
                    tmp_channels,
                    out_channels,
                    kernel_size=kernel_size,
                    stride=stride,
```

```
                    padding=padding,
                )
            )

            # Batch Normalization
            # - Batch normalization is used to normalize the input layer by adjusting and scaling the activations.
            #   It is used to make the model faster and more stable.
            if batch_norm:
                layers.append(nn.BatchNorm2d(num_features=out_channels))

            # ReLU activation
            # - An activation function is used to introduce non-linearity to the output of a neuron
            layers.append(self.relu)

            # Maxpooling layer
            # - Max pooling is a downsampling operation that reduces the dimensionality of the input
            # - Kernel size is the size of the filter
            # - Stride is the step size for the filter
            if max_pool > 1:
                max_pool_stride = config.get("max_pool_stride", 1)
                layers.append(nn.MaxPool2d(kernel_size=max_pool, stride=max_pool_stride))

            # Update input channels for next iteration
            tmp_channels = out_channels

        # Sequential container for convolutional layers
        self.cv_layers = nn.Sequential(*layers)

        # Calculate input size of tensor after convolutional layers
        with torch.no_grad():
            sample_input = torch.randn(
                1, in_channels, image_size, image_size
            )  # create a random sample input tensor
            conv_output = self.cv_layers(sample_input)
            fc_input_features = (
                conv_output.numel()
            )  # calculate the number of elements in the tensor

        # Fully connected layers
        fc_layers_list = []
        for config in net_config["fc_layers"]:
            out_features = config["out_features"]
            dropout_rate = config.get("dropout_rate", 0)
            batch_norm = config.get("batch_norm", False)

            # Fully connected layer
            # - Input features are the number of neurons in previous layer
            # - Output features are the number of neurons to create in current layer
            fc_layers_list.append(
                nn.Linear(in_features=fc_input_features, out_features=out_features)
            )

            # Batch Normalization
            # - Batch normalization is used to normalize the input layer by adjusting and scaling the activations.
            #   It is used to make the model faster and more stable.
            if batch_norm:
                fc_layers_list.append(nn.BatchNorm1d(out_features))

            # ReLU activation
            # - An activation function is used to introduce non-linearity to the output of a neuron
            fc_layers_list.append(self.relu)

            # Dropout
            # - Dropout is a regularization technique to prevent overfitting by randomly setting some output features t
            if dropout_rate > 0:
                fc_layers_list.append(nn.Dropout(p=dropout_rate))

            # Update input features for next iteration
            fc_input_features = out_features

        # Output layer (no dropout or activation)
        fc_layers_list.append(nn.Linear(fc_input_features, num_classes))

        # Sequential container for fully connected layers
        self.fc_layers = nn.Sequential(*fc_layers_list)

    def forward(self, x):
        x = self.cv_layers(x)  # pass through convolutional layers
```

```
        x = x.flatten(
            start_dim=1, end_dim=-1
        )  # flatten tensor from convolutional layers for the linear fully connected layers
        x = self.fc_layers(x)  # pass through fully connected layers
        return x
```

---

**File: train_model.py**

```
from default_config import default_config, batch_size, image_size, val_dir
from loaders import get_train_loader, get_test_loader
import time
import torch
import torch.nn as nn
import torch.optim as optim
from torchinfo import summary


def train_model(model, device, config=default_config):
    label = config["label"]
    n_epochs = config["n_epochs"]

    train_config = config["train_config"]
    transform_config = config["transform_config"]

    model = model.to(device)  # move model to device

    criterion = nn.CrossEntropyLoss()  # loss function

    optimizer_type = train_config["optimizer_type"]
    learning_rate = train_config["learning_rate"]
    weight_decay = train_config.get("weight_decay", 0.0)
    momentum = train_config.get("momentum", 0.0)
    reg_type = train_config["reg_type"]
    reg_lambda = train_config["reg_lambda"]
    step_size = train_config["step_size"]
    gamma = train_config["gamma"]

    # Get transformations
    train_loader = get_train_loader(transform_config)
    val_loader = get_test_loader(val_dir)

    # Select optimizer
    if optimizer_type == "SGD":
        optimizer = optim.SGD(
            model.parameters(), lr=learning_rate, momentum=momentum, weight_decay=weight_decay
        )
    elif optimizer_type == "Adam":
        # If weight decay is specified, apply AdamW instead
        if weight_decay > 0:
            optimizer = optim.AdamW(model.parameters(), lr=learning_rate, weight_decay=weight_decay)
        else:
            optimizer = optim.Adam(model.parameters(), lr=learning_rate)

    # Scheduler
    if step_size is not None and gamma is not None:
        scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=step_size, gamma=gamma)

    train_losses = []
    train_accuracies = []
    val_losses = []
    val_accuracies = []

    print(f"\nExperiment: {label}")

    # Track total training time
    total_start_time = time.time()

    for epoch in range(n_epochs):

        start_time = time.time()

        model.train()  # set model to training mode

        train_loss = 0.0
        correct = 0
        total = 0
```

```python
    # Progress bar for training batches
    for images, labels in train_loader:
        images, labels = images.to(device), labels.to(device)  # move data to device
        optimizer.zero_grad()  # zero the parameter gradients
        outputs = model(images)
        loss = criterion(outputs, labels)  # calculate loss

        # Apply regularization if specified
        if reg_type == "L1":
            l1_norm = sum(param.abs().sum() for param in model.parameters())
            loss += reg_lambda * l1_norm
        elif reg_type == "L2":
            l2_norm = sum(param.pow(2).sum() for param in model.parameters())
            loss += reg_lambda * l2_norm

        loss.backward()  # backpropagation
        optimizer.step()  # update weights
        train_loss += loss.item()  # add the loss to the training set loss

        # Calculate training accuracy
        _, predicted = torch.max(outputs, 1)  # get predicted class
        total += labels.size(0)
        correct += (predicted == labels).sum().item()  # count correct predictions

    if step_size is not None and gamma is not None:
        scheduler.step()  # update learning rate

    # Training set statistics
    train_losses.append(round(train_loss / len(train_loader), 4))
    train_accuracies.append(round(100 * correct / total, 2) if total > 0 else 0)

    # Evaluation on validation set
    model.eval()  # set model to evaluation mode

    val_loss = 0.0
    correct = 0
    total = 0

    with torch.no_grad():  # no need to calculate gradients for validation set
        for images, labels in val_loader:
            images, labels = images.to(device), labels.to(device)  # move data to device
            outputs = model(images)
            loss = criterion(outputs, labels)  # calculate loss
            val_loss += loss.item()  # add the loss to the validation set loss
            _, predicted = torch.max(outputs, 1)  # get predicted class
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    # Validation set statistics
    val_losses.append(round(val_loss / len(val_loader), 4))
    val_accuracies.append(round(100 * correct / total, 2) if total > 0 else 0)

    # Print epoch summary
    epoch_duration = round(time.time() - start_time)
    print(
        f"Epoch {epoch+1}/{n_epochs} | Train Loss: {train_losses[-1]:.4f} (acc. {train_accuracies[-1]:.2f}%) | "
        f"Val Loss: {val_losses[-1]:.4f} (acc. {val_accuracies[-1]:.2f}%) | Time: {epoch_duration}s"
    )

# Calculate and print total training time
total_training_time = round(time.time() - total_start_time)
print(f"Training Time: {total_training_time}s")

# Save model and metrics to file
with open(f"models/{label}.txt", "w") as f:
    model_summary = summary(
        model, input_size=(batch_size, 3, image_size, image_size), verbose=0
    )
    f.write(str(model_summary))
    f.write("\nTraining and Validation Metrics:\n")
    f.write(f"Train Losses: {train_losses}\n")
    f.write(f"Train Accuracies: {train_accuracies}\n")
    f.write(f"Val Losses: {val_losses}\n")
    f.write(f"Val Accuracies: {val_accuracies}\n")

# Save model to file
torch.save(model.state_dict(), f"models/{label}.pth")
```

```python
    return {
        "n_epochs": n_epochs,
        "train_losses": train_losses,
        "val_losses": val_losses,
        "train_accuracies": train_accuracies,
        "val_accuracies": val_accuracies,
    }
```

---

**File: result_handler.py**

```python
from convolutionalNetwork import ConvolutionalNetwork
from train_model import train_model


def result_handler(configs, device):
    results = {}
    for config in configs:
        model = ConvolutionalNetwork(config["net_config"])
        result = train_model(model, device, config)
        label = config["label"]
        results[label] = result
    return results
```

---

**File: plot_scores.py**

```python
import matplotlib.pyplot as plt
from matplotlib.ticker import MaxNLocator


def plot_scores(results):
    for label, data in results.items():
        fig, axes = plt.subplots(1, 2, figsize=(5, 2.5))

        epochs = range(1, data["n_epochs"] + 1)

        # Plot loss
        axes[0].plot(epochs, data["train_losses"], marker="o", markersize=3, label=f"{label} train")
        axes[0].plot(
            epochs,
            data["val_losses"],
            marker="o",
            markersize=3,
            linestyle="--",
            label=f"{label} val",
        )
        axes[0].set_title("loss")
        axes[0].set_xlabel("epoch")
        axes[0].set_ylabel("loss")

        # Plot accuracy
        axes[1].plot(
            epochs, data["train_accuracies"], marker="o", markersize=3, label=f"{label} train"
        )
        axes[1].plot(
            epochs,
            data["val_accuracies"],
            marker="o",
            markersize=3,
            linestyle="--",
            label=f"{label} val",
        )
        axes[1].set_title("accuracy")
        axes[1].set_xlabel("epoch")
        axes[1].set_ylabel("accuracy (%)")

        # Set 4 ticks on the x and y-axis
        axes[0].yaxis.set_major_locator(MaxNLocator(nbins=4))
        axes[1].yaxis.set_major_locator(MaxNLocator(nbins=4))
        axes[0].xaxis.set_major_locator(MaxNLocator(nbins=4))
        axes[1].xaxis.set_major_locator(MaxNLocator(nbins=4))

        # Create a single legend below the plots
        handles, labels = axes[0].get_legend_handles_labels()
        fig.legend(handles, labels, loc="lower center", bbox_to_anchor=(0.5, 0), ncol=2)
```

```
    plt.tight_layout(rect=[0, 0.1, 1, 0.9])
    plt.show()
```

---

**File: predict.py**

```
import torch
import matplotlib.pyplot as plt
from collections import defaultdict
from loaders import get_test_loader
from denormalize_image import denormalize_image
from default_config import label_map, test_dir
import csv


def plot_classified_and_misclassified(correctly_classified, misclassified):
   num_images = 3

   # Sort by confidence
   correctly_classified.sort(key=lambda x: x[1], reverse=True)
   misclassified.sort(key=lambda x: x[1], reverse=True)

   # Get top `num_images` for each class
   top_correct = []
   top_incorrect = []
   for label in set(x[2] for x in correctly_classified):
       top_correct.extend([x for x in correctly_classified if x[2] == label][:num_images])
       top_incorrect.extend([x for x in misclassified if x[2] == label][:num_images])

   # Plot correctly and misclassified images
   fig, axs = plt.subplots(2, num_images * 2 + 1, figsize=(25, 10))

   # Add row labels
   axs[0, 0].text(
       0.5, 0.5, "Correctly Classified", fontsize=12, ha="center", va="center", rotation=90
   )
   axs[0, 0].axis("off")
   axs[1, 0].text(0.5, 0.5, "Misclassified", fontsize=12, ha="center", va="center", rotation=90)
   axs[1, 0].axis("off")

   # Show correctly classified images
   for i, (img, prob, true_label, pred_label) in enumerate(top_correct):
       img = denormalize_image(img)
       axs[0, i + 1].imshow(img.permute(1, 2, 0))
       axs[0, i + 1].set_title(f"True: {true_label}, Pred: {pred_label}, Prob: {prob*100:.2f}%")
       axs[0, i + 1].axis("off")

   # Show misclassified images
   for i, (img, prob, true_label, pred_label) in enumerate(top_incorrect):
       img = denormalize_image(img)
       axs[1, i + 1].imshow(img.permute(1, 2, 0))
       axs[1, i + 1].set_title(f"True: {true_label}, Pred: {pred_label}, Prob: {prob*100:.2f}%")
       axs[1, i + 1].axis("off")

   plt.tight_layout()
   plt.show()


def predict(model, device, model_path, results_file="image_probabilities.csv"):
   # Load model
   model.to(device)
   model.load_state_dict(torch.load(model_path, weights_only=True))
   model.eval()

   # Get loader
   test_loader = get_test_loader(test_dir)

   misclassified = []
   correctly_classified = []
   correct_class_counts = defaultdict(int)
   misclass_class_counts = defaultdict(int)

   # Access image paths directly from the dataset within the loader
   image_paths = [sample[0] for sample in test_loader.dataset.samples]

   # Open CSV file for writing
```

```python
    with open(results_file, mode="w", newline="") as file:
        writer = csv.writer(file)
        writer.writerow(
            ["Image Name", "Correct Prediction", "True Label", "Predicted Label", "Probabilities"]
        )

        with torch.no_grad():
            for batch_idx, (images, labels) in enumerate(test_loader):
                images, labels = images.to(device), labels.to(device)
                outputs = model(images)

                probs = torch.softmax(outputs, dim=1)
                predicted_labels = torch.argmax(probs, dim=1)

                for idx in range(images.size(0)):
                    image_index = batch_idx * test_loader.batch_size + idx
                    img = images[idx].cpu()
                    image_name = image_paths[image_index]

                    true_label_num = labels[idx].cpu().item()
                    pred_label_num = predicted_labels[idx].cpu().item()
                    true_label = label_map[true_label_num]
                    pred_label = label_map[pred_label_num]

                    prob_values = probs[idx].cpu().tolist()
                    correct_prediction = true_label == pred_label

                    if correct_prediction:
                        correctly_classified.append((img, max(prob_values), true_label, pred_label))
                        correct_class_counts[true_label] += 1
                    else:
                        misclassified.append((img, max(prob_values), true_label, pred_label))
                        misclass_class_counts[true_label] += 1

                    # Write to CSV
                    writer.writerow(
                        [
                            image_name,
                            correct_prediction,
                            true_label,
                            pred_label,
                            [f"{p*100:.2f}%" for p in prob_values],
                        ]
                    )

    # Print total counts and accuracy
    total_correct = len(correctly_classified)
    total_misclassified = len(misclassified)
    total = total_correct + total_misclassified
    accuracy = round((total_correct / total) * 100, 2)
    total_cats = correct_class_counts["cat"] + misclass_class_counts["cat"]
    total_dogs = correct_class_counts["dog"] + misclass_class_counts["dog"]
    cat_acc = round(correct_class_counts["cat"] / total_cats * 100, 2)
    dog_acc = round(correct_class_counts["dog"] / total_dogs * 100, 2)
    print(
        f"Total Correctly Classified: {total_correct} | Total Misclassified: {total_misclassified} | Accuracy: {accurac
    )

    # Print class-wise statistics
    print("Class-wise Correctly Classified Counts:")
    for label, count in correct_class_counts.items():
        print(f"- Class {label}: {count}")
    print("Class-wise Misclassified Counts:")
    for label, count in misclass_class_counts.items():
        print(f"- Class {label}: {count}")
    print(f"Cat Accuracy: {cat_acc}% | Dog Accuracy: {dog_acc}%")

    plot_classified_and_misclassified(correctly_classified, misclassified)
```

---

**File: plot_individual_feature_maps.py**

```python
import matplotlib.pyplot as plt
import torch
from PIL import Image
from loaders import get_test_transform
```

```python
def plot_individual_feature_maps(
    model, device, img_path, output_file="individual_feature_maps.png"
):
    # Load and preprocess the image
    img = Image.open(img_path).convert("RGB")
    test_transform = get_test_transform()
    img_tensor = (
        test_transform(img).unsqueeze(0).to(device)
    )  # apply transforms and add batch dimension

    # Plot and save the original image
    _, ax = plt.subplots(figsize=(5, 5))
    ax.imshow(img)
    ax.axis("off")
    ax.set_title("Original Image", fontsize=14)
    original_image_file = f"{output_file}_original.png"
    plt.savefig(original_image_file)
    plt.show()

    # Set model to evaluation mode and move to correct device
    model.eval()
    model.to(device)

    # Collect all convolutional layers
    conv_layers = []
    model_children = list(model.children())

    for layer in model_children:
        if isinstance(layer, torch.nn.Conv2d):
            conv_layers.append(layer)
        elif isinstance(layer, torch.nn.Sequential):
            for child in layer.children():
                if isinstance(child, torch.nn.Conv2d):
                    conv_layers.append(child)

    # Forward pass through model and collect feature maps
    outputs = []
    for layer in conv_layers:
        img_tensor = layer(img_tensor)
        outputs.append(img_tensor)

    # Create the figure for feature maps
    for layer_idx, feature_map in enumerate(outputs):
        feature_map = feature_map.squeeze(0).cpu()  # remove batch dimension and move to CPU
        num_filters = feature_map.size(0)  # number of filters in this layer

        # Determine grid layout for plotting all filters
        col_size = 8  # number of columns
        row_size = (num_filters + col_size - 1) // col_size

        _, axes = plt.subplots(row_size, col_size, figsize=(col_size * 2, row_size * 2))
        axes = axes.flatten()

        # Plot each filter's feature map
        for filter_idx in range(num_filters):
            single_filter_map = feature_map[filter_idx].detach().numpy()
            axes[filter_idx].imshow(single_filter_map, cmap="gray")
            axes[filter_idx].axis("off")
            axes[filter_idx].set_title(f"Filter {filter_idx + 1}", fontsize=8)

        # Turn off any unused subplots
        for idx in range(num_filters, len(axes)):
            axes[idx].axis("off")

        # Save figure for each layer
        plt.tight_layout()
        layer_output_file = f"{output_file}_layer_{layer_idx + 1}.png"
        plt.savefig(layer_output_file)
        plt.show()
```

---

**File: plot_transformations.py**

```python
import matplotlib.pyplot as plt
import math
```

```python
num_images_in_row = 3


def plot_transformations(transformed_images, transforms):
    num_images = len(transformed_images)
    rows = math.ceil(num_images / num_images_in_row)

    _, axes = plt.subplots(
        rows,
        min(num_images_in_row, num_images),
        figsize=(3 * min(num_images_in_row, num_images), 3 * rows),
    )

    if rows == 1:
        axes = [axes]
    elif num_images <= num_images_in_row:
        axes = [axes]

    # Plot each image in the corresponding subplot
    for i, img in enumerate(transformed_images):
        row, col = divmod(i, num_images_in_row)
        axes[row][col].imshow(img)
        axes[row][col].set_title(f"{transforms[i]}")

    # Turn off axis for all subplots
    for row in axes:
        for ax in row:
            ax.axis("off")

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