Курсовой проект

Transfer learning

Выполнил Лисин Роман, М8О-406Б-20

Подробно изучить туториал и написать комментарий к каждой строчке кода (https://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html)

```
import torch
import torch.nn as nn
import torch.optim as optim
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim import lr_scheduler
import torch.backends.cudnn as cudnn
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
from PIL import Image
from tempfile import TemporaryDirectory

# Set cuDNN benchmark mode for faster training
cudnn.benchmark = True

# Set matplotlib to interactive mode for faster plotting
plt.ion()

# Data augmentation and normalization for training, and just
normalization for validation
data_transforms = {
    'train': transforms.Compose([
        # Randomly crop the image to 224x224
        transforms.RandomResizedCrop(224),
        # Randomly flip the image horizontally
        transforms.RandomHorizontalFlip(),
        # Convert the image to a tensor
        transforms.ToTensor(),
        # Normalize the image using the mean and standard deviation of
```

```
the ImageNet dataset
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
    'val': transforms.Compose([
        # Resize the image to 256x256
        transforms.Resize(256),
        # Crop the center 224x224 region of the image
        transforms.CenterCrop(224),
        # Convert the image to a tensor
        transforms.ToTensor(),
        # Normalize the image using the mean and standard deviation of
the ImageNet dataset
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224,
0.225])
    ]),
}

# Define the data directory
data_dir = 'data/hymenoptera_data'

# Create the ImageFolder datasets for training and validation
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}

# Create the DataLoader for training and validation
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x],
batch_size=4,
                                              shuffle=True,
num_workers=4)
              for x in ['train', 'val']}

# Get the sizes of the datasets
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}

# Get the class names
class_names = image_datasets['train'].classes

# Set the device to use for training
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")


# Define a function to show an image
def imshow(inp, title=None):
    """Display image for Tensor."""
    # Convert the image from a tensor to a numpy array
    inp = inp.numpy().transpose((1, 2, 0))
    # Un-normalize the image using the mean and standard deviation of
```

```
the ImageNet dataset
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    # Clip the image to be between 0 and 1
    inp = np.clip(inp, 0, 1)
    # Plot the image
    plt.imshow(inp)
    if title is not None:
        plt.title(title)
    plt.pause(0.001)  # pause a bit so that plots are updated


# Get a batch of training data
inputs, classes = next(iter(dataloaders['train']))

# Make a grid from the batch
out = torchvision.utils.make_grid(inputs)

# Show the grid
imshow(out, title=[class_names[x] for x in classes])


def train_model(model, criterion, optimizer, scheduler,
num_epochs=25):

    # Record the time when training starts
    since = time.time()

    # Create a temporary directory to save training checkpoints
    with TemporaryDirectory() as tempdir:
        # Create a path to save the best model parameters
        best_model_params_path = os.path.join(tempdir,
'best_model_params.pt')

        # Save the initial model parameters
        torch.save(model.state_dict(), best_model_params_path)
        # Initialize the best accuracy to 0
        best_acc = 0.0

        # Iterate over the specified number of epochs
        for epoch in range(num_epochs):
            # Print the current epoch and number of epochs
            print(f'Epoch {epoch}/{num_epochs - 1}')
            print('-' * 10)

            # Each epoch has a training and validation phase
            for phase in ['train', 'val']:
                if phase == 'train':
                    model.train()  # Set model to training mode
```

```
                else:
                    model.eval()   # Set model to evaluate mode

                # Initialize the running loss and number of correct
predictions
                running_loss = 0.0
                running_corrects = 0

                # Iterate over the data.
                for inputs, labels in dataloaders[phase]:
                    # Move the inputs and labels to the GPU
                    inputs = inputs.to(device)
                    labels = labels.to(device)

                    # Zero the parameter gradients
                    optimizer.zero_grad()

                    # Forward pass
                    # Track history if only in train
                    with torch.set_grad_enabled(phase == 'train'):
                        # Compute the outputs
                        outputs = model(inputs)
                        # Get the predicted labels
                        _, preds = torch.max(outputs, 1)
                        # Compute the loss
                        loss = criterion(outputs, labels)

                        # Backward pass and optimize only if in
training phase
                        if phase == 'train':
                            # Compute the gradients of the loss with
respect to the model parameters
                            loss.backward()
                            # Update the model parameters
                            optimizer.step()

                    # Update the running loss and number of correct
predictions
                    running_loss += loss.item() * inputs.size(0)
                    running_corrects += torch.sum(preds ==
labels.data)
                if phase == 'train':
                    scheduler.step()

                # Compute the epoch loss and accuracy
                epoch_loss = running_loss / dataset_sizes[phase]
                epoch_acc = running_corrects.double() /
dataset_sizes[phase]

                # Print the epoch loss and accuracy
```

```
                print(f'{phase} Loss: {epoch_loss:.4f} Acc:
{epoch_acc:.4f}')

                # Check if the current model is better than the best
model so far
                if phase == 'val' and epoch_acc > best_acc:
                    # Update the best accuracy
                    best_acc = epoch_acc
                    # Save the best model parameters
                    torch.save(model.state_dict(),
best_model_params_path)

            # Print a newline
            print()

        # Compute the total training time
        time_elapsed = time.time() - since
        print(f'Training complete in {time_elapsed // 60:.0f}m
{time_elapsed % 60:.0f}s')
        print(f'Best val Acc: {best_acc:4f}')

        # Load the best model weights
        model.load_state_dict(torch.load(best_model_params_path))
    # Return the trained model
    return model


# Define a function to visualize the model's predictions
def visualize_model(model, num_images=6):
    # Set the model to evaluation mode
    was_training = model.training
    model.eval()

    # Initialize the number of images shown
    images_so_far = 0

    # Create a figure
    fig = plt.figure()

    # Iterate over the validation data
    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            # Move the inputs and labels to the device
            inputs = inputs.to(device)
            labels = labels.to(device)

            # Forward pass the inputs through the model
            outputs = model(inputs)

            # Get the predicted class labels
```

```
            _, preds = torch.max(outputs, 1)

            # Iterate over the batch size
            for j in range(inputs.size()[0]):
                # Increment the number of images shown
                images_so_far += 1

                # Create a subplot
                ax = plt.subplot(num_images//2, 2, images_so_far)

                # Turn off the axes
                ax.axis('off')

                # Set the title of the subplot
                ax.set_title(f'predicted: {class_names[preds[j]]}')

                # Show the image
                imshow(inputs.cpu().data[j])

                # If the number of images shown is equal to the
specified number of images, break out of the loop
                if images_so_far == num_images:
                    # Set the model back to training mode
                    model.train(mode=was_training)

                    # Return
                    return

        # Set the model back to training mode
        model.train(mode=was_training)


# Load a pre-trained ResNet-18 model
model_ft = models.resnet18(weights='IMAGENET1K_V1')

# Get the number of features in the last fully connected layer
num_ftrs = model_ft.fc.in_features

# Replace the last fully connected layer with a new one with the
correct number of outputs
model_ft.fc = nn.Linear(num_ftrs, 2)

# Move the model to the device
model_ft = model_ft.to(device)

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr=0.001,
momentum=0.9)
```

```
# Define the learning rate scheduler
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size=7,
gamma=0.1)

# Train the model
model_ft = train_model(model_ft, criterion, optimizer_ft,
exp_lr_scheduler,
                       num_epochs=25)

# Visualize the model's predictions
visualize_model(model_ft)


# Load a pre-trained ResNet-18 model
model_conv = torchvision.models.resnet18(weights='IMAGENET1K_V1')

# Freeze the parameters of the pre-trained model
for param in model_conv.parameters():
    param.requires_grad = False

# Get the number of features in the last fully connected layer
num_ftrs = model_conv.fc.in_features

# Replace the last fully connected layer with a new one with the
correct number of outputs
model_conv.fc = nn.Linear(num_ftrs, 2)

# Move the model to the device
model_conv = model_conv.to(device)

# Define the loss function
criterion = nn.CrossEntropyLoss()

# Observe that only parameters of final layer are being optimized as
opposed to before.
optimizer_conv = optim.SGD(model_conv.fc.parameters(), lr=0.001,
momentum=0.9)

# Define the learning rate scheduler
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size=7,
gamma=0.1)

# Train the model
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=25)

# Visualize the model's predictions
visualize_model(model_conv)

# Turn off interactive mode for matplotlib
```

```
plt.ioff()

# Show the plots
plt.show()


# Define a function to visualize the model's predictions on a specific
image
def visualize_model_predictions(model,img_path):
    # Set the model to evaluation mode
    was_training = model.training
    model.eval()

    # Load the image
    img = Image.open(img_path)

    # Preprocess the image
    img = data_transforms['val'](img)

    # Add a batch dimension to the image
    img = img.unsqueeze(0)

    # Move the image to the device
    img = img.to(device)

    # Forward pass the image through the model
    with torch.no_grad():
        outputs = model(img)

    # Get the predicted class label
    _, preds = torch.max(outputs, 1)

    # Create a figure
    fig = plt.figure()

    # Create a subplot
    ax = plt.subplot(2,2,1)

    # Turn off the axes
    ax.axis('off')

    # Set the title of the subplot
    ax.set_title(f'Predicted: {class_names[preds[0]]}')

    # Show the image
    imshow(img.cpu().data[0])

    # Set the model back to training mode
    model.train(mode=was_training)
```

```
# Visualize the model's predictions on a specific image
visualize_model_predictions(
    model_conv,
    img_path='data/hymenoptera_data/val/bees/72100438_73de9f17af.jpg'
)

# Turn off interactive mode for matplotlib
plt.ioff()

# Show the plots
plt.show()
```

Выбрать наборы данных и обосновать его выбор

В качестве набора данных для обучения и тестирования будем использовать датасет Fashion MNIST. Это база данных изображений товаров Zalando, состоящая из обучающего набора из 60 000 примеров и тестового набора из 10 000 примеров. Каждый пример - это изображение размером 28x28 градаций серого, связанное с меткой из 10 классов.

С помощью данного датасета можно подготовить модель для решения реальной задачи классификации вещей для интернет-магазинов одежды. Например, если покупатель захочет подобрать вещь по картинке, то с помощью предсказанного класса вещи пользователю можно предложить доступные товары из этой категории.

Выбрать метрики качества и обосновать их выбор

Для оценки качества решения задачи достаточно использовать стандартные метрики - accuracy, loss. Accuracy показывает долю правильно классифицированных вещей, что нужно для данной практической задачи. Loss помогает избежать переобучения и недообучения.

Создание бейзлайна и оценка качества

```
import torch
from torchvision import datasets, transforms
from torchvision import models
import torch.optim as optim
```

Загружаем датасет Fashion MNIST.

```
train_dataset = datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=transforms.ToTensor()
)

test_dataset = datasets.FashionMNIST(
    root='./data',
```

```
    train=False,
    download=True,
    transform=transforms.ToTensor()
)

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
shuffle=False)
```

Подсчёт метрик accuracy и loss.

```
def accuracy_and_loss(model, data_loader, device="cuda"):
    total = 0
    correct = 0
    loss_total = 0

    with torch.no_grad():
        for data in data_loader:
            images, labels = data[0].to(device), data[1].to(device)

            outputs = model(images)
            loss = torch.nn.CrossEntropyLoss()(outputs, labels)
            _, predicted = torch.max(outputs.data, 1)
            loss_total += loss.item()
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    return 100 * correct / total, loss_total / len(data_loader)
```

Используем модель ResNet из pytorch для классификации вещей. Добавляем свёрточный слой, так как изображения вещей чёрно-белые, а ResNet изначально заточена под цветные изображения.

```
if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

model = models.resnet18()
model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=7, stride=2,
padding=3, bias=False)
model.fc = torch.nn.Linear(model.fc.in_features, 10)
model.to(device)
```

```
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
```

```
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)
```

Обучение ResNet.

```
from tqdm import tqdm

optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    total_loss = 0
    progress = tqdm(enumerate(train_loader), desc="Loss: ")
    model.train()
    for i, data in enumerate(train_loader):
        images, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = torch.nn.CrossEntropyLoss()(outputs, labels)
        loss.backward()
        optimizer.step()
        cur_loss = loss.item()
        total_loss += cur_loss
        if i % 100 == 0:
          progress.set_description("Loss:
{:.4f}".format(total_loss/(i+1)))
    print(f"\nEpoch {epoch+1}/5, training loss:
{total_loss/len(train_loader)}")
```

```
Loss: 0.2827: : 0it [01:06, ?it/s]
Loss: 0.4365: : 0it [00:22, ?it/s]


Epoch 1/5, training loss: 0.43204232379158675


Loss: 0.4365: : 0it [00:23, ?it/s]

Loss: 0.3285: : 0it [00:00, ?it/s]


Epoch 2/5, training loss: 0.3032578127796271

Loss: 0.3042: : 0it [00:23, ?it/s]
Loss: 0.2681: : 0it [00:22, ?it/s]


Epoch 3/5, training loss: 0.2684093166166531


Loss: : 0it [00:00, ?it/s]


Epoch 4/5, training loss: 0.2437274051802372

Loss: 0.2436: : 0it [00:23, ?it/s]
Loss: 0.2203: : 0it [00:22, ?it/s]


Epoch 5/5, training loss: 0.21942233395522465
```

Подсчёт метрик.

```
test_acc, test_loss = accuracy_and_loss(model, test_loader)

print(f'Test Accuracy: {test_acc:.2f}%, Test Loss: {test_loss:.4f}')

Test Accuracy: 89.94%, Test Loss: 0.2749
```

Улучшение бейзлайна

Добавим аугментацию данных. Поменяем размер изображений и нормализуем их.

```
fashion_mnist = datasets.FashionMNIST(download=False,
root="./data").data.float()
transformations = transforms.Compose([transforms.Resize((224, 224)),
                                      transforms.ToTensor(),

transforms.Normalize((fashion_mnist.mean()/255,),
(fashion_mnist.std()/255,))])
```

Далее снова обучим модель.

```
train_dataset = datasets.FashionMNIST(
    root='./data',
    train=True,
    download=True,
    transform=transformations
)

test_dataset = datasets.FashionMNIST(
    root='./data',
    train=False,
    download=True,
    transform=transformations
)

train_loader = torch.utils.data.DataLoader(train_dataset,
batch_size=64, shuffle=True)
test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
shuffle=False)

if torch.cuda.is_available():
    device = torch.device("cuda")
else:
    device = torch.device("cpu")

model = models.resnet18()
model.conv1 = torch.nn.Conv2d(1, 64, kernel_size=7, stride=2,
padding=3, bias=False)
model.fc = torch.nn.Linear(model.fc.in_features, 10)
model.to(device)
```

```
ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
```

```
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer4): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2),
bias=False)
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=512, out_features=10, bias=True)
)

optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    total_loss = 0
    progress = tqdm(enumerate(train_loader), desc="Loss: ")
    model.train()
    for i, data in enumerate(train_loader):
        images, labels = data[0].to(device), data[1].to(device)
        optimizer.zero_grad()
        outputs = model(images)
        loss = torch.nn.CrossEntropyLoss()(outputs, labels)
        loss.backward()
        optimizer.step()
        cur_loss = loss.item()
        total_loss += cur_loss
        if i % 100 == 0:
          progress.set_description("Loss:
{:.4f}".format(total_loss/(i+1)))
    print(f"\nEpoch {epoch+1}/5, training loss:
{total_loss/len(train_loader)}")


Loss: 0.2203: : 0it [15:07, ?it/s]

Loss: 2.3946: : 0it [00:00, ?it/s]


Epoch 1/5, training loss: 0.41054935562712297

Loss: 0.4160: : 0it [03:22, ?it/s]
Loss: 0.2589: : 0it [03:12, ?it/s]


Epoch 2/5, training loss: 0.25847844969330314


Loss: 0.2589: : 0it [03:20, ?it/s]

Loss: 0.1894: : 0it [00:00, ?it/s]


Epoch 3/5, training loss: 0.21289755486206077
```

```
Loss: 0.2137: : 0it [03:29, ?it/s]
Loss: 0.1879: : 0it [03:17, ?it/s]


Epoch 4/5, training loss: 0.18820305226215803


Loss: 0.1879: : 0it [03:25, ?it/s]

Loss: 0.1299: : 0it [00:00, ?it/s]


Epoch 5/5, training loss: 0.16643963331408274

test_acc, test_loss = accuracy_and_loss(model, test_loader)

print(f'Test Accuracy: {test_acc:.2f}%, Test Loss: {test_loss:.4f}')

Test Accuracy: 91.93%, Test Loss: 0.2238
```

Гипотеза с аугментацией данных оказалась верна, так как удалось увеличить accuracy с 89.94% до 91.93% с её помощью.

Архитектура сверточной нейронной сети ResNet

ResNet (Residual Network) - это тип сверточной нейронной сети (CNN), разработанный для решения проблемы деградации градиента, которая часто возникает при обучении очень глубоких нейронных сетей. ResNet использует архитектуру с остаточными блоками, которые позволяют сети учиться на своих ошибках и улучшать свою производительность по мере обучения.

Архитектура ResNet состоит из следующих основных компонентов:

Сверточные слои: ResNet использует сверточные слои для извлечения признаков из входных данных. Эти слои применяют фильтры к входным данным, чтобы обнаружить шаблоны и особенности.

Слои активации: После каждого сверточного слоя добавляется слой активации, такой как ReLU, чтобы ввести нелинейность в сеть. Это позволяет сети моделировать сложные функции.

Батч-нормализация: ResNet также использует батч-нормализацию для стабилизации обучения и ускорения сходимости. Батч-нормализация нормализует входные данные по батчам, что делает их менее чувствительными к колебаниям распределения данных.

Остаточные блоки: Основным строительным блоком ResNet является остаточный блок. Остаточный блок состоит из двух сверточных слоев, за которыми следуют слои активации и батч-нормализации. Выход остаточного блока суммируется с входными данными, что позволяет сети учиться на разнице между входными и выходными данными.

Глобальный средний пулинг: После последнего остаточного блока используется глобальный средний пулинг для сведения размерности выходных данных. Это приводит к

вектору признаков фиксированной длины, который затем подается на полностью связанный слой для классификации.

Остаточные блоки играют решающую роль в архитектуре ResNet. Они позволяют сети обучаться быстрее и достигать более высокой точности, решая проблему деградации градиента. Деградация градиента возникает в глубоких сетях, когда градиенты становятся очень малыми по мере прохождения через сеть, что затрудняет обучение более глубоких слоев.

Остаточные блоки облегчают обратное распространение градиентов, поскольку градиенты могут проходить через них напрямую, минуя нелинейности. Это обеспечивает более эффективный поток градиентов и позволяет сети учиться на своих ошибках.

Преимущества ResNet

Более высокая точность: ResNet достигает более высокой точности по сравнению с другими архитектурами CNN на различных задачах классификации изображений.

Быстрая сходимость: Остаточные блоки позволяют ResNet сходиться быстрее, что сокращает время обучения.

Меньшая склонность к переобучению: ResNet менее подвержен переобучению, чем другие архитектуры CNN.

Возможность обучения на больших данных: ResNet может быть эффективно обучен на больших наборах данных с миллионами изображений.

Недостатки ResNet

Высокие вычислительные затраты: ResNet требует больших вычислительных ресурсов для обучения из-за большого количества слоев.

Большой размер модели: ResNet имеет больший размер модели по сравнению с другими архитектурами CNN.

Сложность настройки: Настройка ResNet для новых задач может быть более сложной из-за большого количества гиперпараметров.

Вариации ResNet

Существует несколько вариаций ResNet, разработанных для различных задач и наборов данных. Вот некоторые распространенные варианты:

ResNet-18: Состоит из 18 остаточных блоков и используется для небольших наборов данных.

ResNet-50: Состоит из 50 остаточных блоков и используется для средних и больших наборов данных.

ResNet-101: Состоит из 101 остаточного блока и используется для очень больших наборов данных.

ResNet-152: Состоит из 152 остаточных блоков и используется для самых больших наборов данных.

Приложения ResNet

ResNet широко используется в различных приложениях, включая:

Классификация изображений: ResNet является одной из наиболее популярных архитектур для классификации изображений и используется в таких задачах, как распознавание объектов, распознавание лиц и медицинская диагностика.

Обнаружение объектов: ResNet используется для обнаружения объектов на изображениях и используется в таких приложениях, как автомобили с автоматическим управлением и робототехника.

Сегментация изображений: ResNet используется для сегментации изображений на различные объекты или области и используется в таких приложениях, как медицинская визуализация и автономное вождение.

Генерация изображений: ResNet также используется для генерации изображений, например, в приложениях увеличения разрешения и создания изображений на основе текста.

Реализация нейросетевой архитектуры

```
import torch
import torchvision
import torch.nn as nn

class ResNet(nn.Module):
    def __init__(self):
        super(ResNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 64, kernel_size=7, stride=2,
padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2,
padding=1)

        self.res_blocks = nn.Sequential(
            ResBlock(64, 64),
            ResBlock(64, 64),
            ResBlock(64, 128),
            ResBlock(128, 128),
            ResBlock(128, 256),
            ResBlock(256, 256),
        )

        self.fc = nn.Linear(256, 10)

    def forward(self, x):
        x = self.conv1(x)
```

```
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        x = self.res_blocks(x)

        x = x.mean(dim=[2, 3])
        x = self.fc(x)

        return x


class ResBlock(nn.Module):
    def __init__(self, in_channels, out_channels):
        super(ResBlock, self).__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(out_channels, out_channels,
kernel_size=3, stride=1, padding=1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        if in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, kernel_size=1,
stride=1),
                nn.BatchNorm2d(out_channels)
            )
        else:
            self.downsample = None

    def forward(self, x):
        residual = x

        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.conv2(x)
        x = self.bn2(x)

        if self.downsample is not None:
            residual = self.downsample(residual)

        x += residual
        x = self.relu(x)

        return x
```

```
model = ResNet()
model.to(device)

ResNet(
  (conv1): Conv2d(1, 64, kernel_size=(7, 7), stride=(2, 2),
padding=(3, 3))
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
ceil_mode=False)
  (res_blocks): Sequential(
    (0): ResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (1): ResBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): ResBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(64, 128, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

```
    (3): ResBlock(
      (conv1): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (4): ResBlock(
      (conv1): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): Conv2d(128, 256, kernel_size=(1, 1), stride=(1, 1))
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (5): ResBlock(
      (conv1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1),
padding=(1, 1))
      (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (fc): Linear(in_features=256, out_features=10, bias=True)
)

optimizer = optim.Adam(model.parameters(), lr=0.001)

for epoch in range(5):
    total_loss = 0
    progress = tqdm(enumerate(train_loader), desc="Loss: ")
    model.train()
    for i, data in enumerate(train_loader):
        images, labels = data[0].to(device), data[1].to(device)
```

```
        optimizer.zero_grad()
        outputs = model(images)
        loss = torch.nn.CrossEntropyLoss()(outputs, labels)
        loss.backward()
        optimizer.step()
        cur_loss = loss.item()
        total_loss += cur_loss
        if i % 100 == 0:
          progress.set_description("Loss:
{:.4f}".format(total_loss/(i+1)))
    print(f"\nEpoch {epoch+1}/5, training loss:
{total_loss/len(train_loader)}")
```

```
Loss: 0.1650: : 0it [09:00, ?it/s]
Loss: 0.6521: : 0it [08:38, ?it/s]


Epoch 1/5, training loss: 0.643514930439402


Loss: 0.6521: : 0it [08:59, ?it/s]

Loss: 0.3994: : 0it [00:00, ?it/s]


Epoch 2/5, training loss: 0.3940116990603872

Loss: 0.3961: : 0it [08:58, ?it/s]
Loss: 0.3294: : 0it [08:37, ?it/s]


Epoch 3/5, training loss: 0.3292567003955211


Loss: 0.3294: : 0it [08:58, ?it/s]

Loss: 0.3203: : 0it [00:00, ?it/s]


Epoch 4/5, training loss: 0.295149716487063

Loss: 0.2957: : 0it [08:58, ?it/s]
Loss: 0.2733: : 0it [08:43, ?it/s]


Epoch 5/5, training loss: 0.2727414288722884

test_acc, test_loss = accuracy_and_loss(model, test_loader)

print(f'Test Accuracy: {test_acc:.2f}%, Test Loss: {test_loss:.4f}')

Test Accuracy: 89.36%, Test Loss: 0.2992
```

Модель ResNet, которую я реализовал самостоятельно, показала accuracy 89.36%. Данный результат хуже, чем у предыдущей модели ResNet из pytorch с accuracy 91.93%. Таким образом, лучшим бейзлайном является ResNet из pytorch с использованием аугментации изображений.