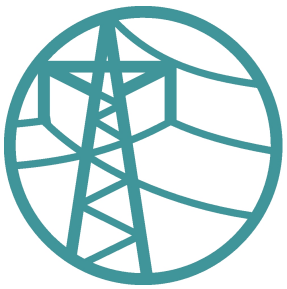

WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Daniel Kleczyński

Wpoprowadzenie do PyTorch: Trenowanie Modeli na Zbiorze Caltech101 i Segmentacja Obrazów z DeepLabV3

Rzeszów, 2024

Spis treści

1. Wprowadzenie

W tej instrukcji omówimy kluczowe funkcje i klasy zawarte w kodzie do trenowania i oceny modeli na zbiorze danych Caltech101. Instrukcja ma na celu naukę PyTorcha poprzez szczegółowe wyjaśnienia i przykłady.

2. Używanie Google Colab z GPU

Aby móc używać Google Colab do trenowania modeli z użyciem GPU, należy wykonać następujące kroki:

2.1. Otwieranie linku w Google Colab

Najpierw otwórz przygotowany notebook w Google Colab, klikając na poniższy link:

- `https://colab.research.google.com/drive/1U9NAjeRkjN5ckB1qt5zH1H73etT4L2Rw?usp=sharing`

2.2. Logowanie się do Google Colab

Po otwarciu linku, jeśli nie jesteś jeszcze zalogowany, wykonaj poniższe kroki:

1) Na stronie logowania wprowadź swoje dane do konta Google.
2) Kliknij `Dalej`, aby się zalogować.

2.3. Zmiana urządzenia wykonawczego na GPU

Aby przyspieszyć trenowanie modeli, zmień urządzenie wykonawcze na GPU:

1) W Google Colab, kliknij `Runtime` (Środowisko wykonawcze) w menu górnym.
2) Wybierz `Change runtime type` (Zmień typ środowiska wykonawczego).
3) W polu `Hardware accelerator` (Akcelerator sprzętowy) wybierz `GPU`.
4) Kliknij `Save` (Zapisz), aby zapisać ustawienia.

Po wykonaniu tych kroków będziesz gotowy do używania Google Colab z GPU do trenowania modeli w PyTorch.

Po wykonaniu tych kroków będziesz gotowy do używania Google Colab z GPU do trenowania modeli w PyTorch.

3. Importowanie bibliotek

Na początku importujemy wszystkie niezbędne biblioteki do przetwarzania obrazów, trenowania modeli, ładowania danych oraz interakcji z modelem za pomocą Gradio.

```
import torch
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import Caltech101
from torchvision.models import resnet18
import torch.nn as nn
import torch.optim as optim
import gradio as gr
```

4. Definiowanie transformacji i ładowanie zbioru danych

4.1. Transformacje

Definiujemy transformacje, które będą stosowane do obrazów wejściowych. Transformacje obejmują:

- `Resize`: Zmiana rozmiaru obrazów do 224x224 pikseli.
- `Grayscale`: Konwersja obrazów na 3-kanałowe obrazy w odcieniach szarości.
- `ToTensor`: Konwersja obrazów do formatu tensora.
- `Normalize`: Normalizacja obrazów z użyciem określonych średnich i odchyleń standardowych.

```
# Define transformations
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # Resize images to 224x224
    transforms.Grayscale(num_output_channels=3),  # Convert
   grayscale images to 3 channels
    transforms.ToTensor(),  # Convert images to tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])  # Normalize images
])
```

4.2. Ładowanie zbioru danych

Ładujemy zbiór danych Caltech101 i dzielimy go na zestawy treningowe i walidacyjne w stosunku 80:20.

```
# Load the dataset
dataset = Caltech101(root='./data', download=True, transform=
    transform)

# Split dataset into training and validation sets
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size,
    val_size])

train_loader = DataLoader(train_dataset, batch_size=32, shuffle=
    True)
val_loader = DataLoader(val_dataset, batch_size=32, shuffle=
    False)
```

5. Definiowanie niestandardowego modelu CNN

Tworzymy niestandardowy model CNN, który będzie wykorzystywany do klasyfikacji obrazów. Model ten składa się z:

- **Części ekstrakcji cech**: Dwie warstwy konwolucyjne, każda z warstwą aktywacji ReLU i warstwą MaxPool.
- **Części klasyfikacji**: Dwie warstwy w pełni połączone z warstwami Dropout i ReLU.

```
class CustomCNN(nn.Module):
    def __init__(self, num_classes):
        super(CustomCNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1)
    ,
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding
    =1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
        )
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(128 * 56 * 56, 256),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(256, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x
```

6. Ładowanie modelu ResNet-18

Załadujemy wstępnie wytrenowany model ResNet-18 i zmodyfikujemy jego ostatnią w pełni połączoną warstwę, aby dopasować ją do liczby klas w zbiorze danych Caltech101.

```
# Load the pre-trained ResNet-18 model
resnet_model = resnet18(pretrained=True)

# Modify the final fully connected layer to match the number of
   classes in Caltech101
num_classes = len(dataset.categories)
resnet_model.fc = nn.Linear(resnet_model.fc.in_features,
   num_classes)
```

7. Inicjalizacja niestandardowego modelu

Inicjalizujemy niestandardowy model z odpowiednią liczbą klas.

```
# Initialize the custom model
custom_model = CustomCNN(num_classes=num_classes)
```

8. Definiowanie funkcji straty i optymalizatorów

Zdefiniujemy funkcję straty (`CrossEntropyLoss`) oraz optymalizatory (`SGD`) dla obu modeli.

```
# Define loss function and optimizer for both models
criterion = nn.CrossEntropyLoss()
resnet_optimizer = optim.SGD(resnet_model.parameters(), lr
   =0.001, momentum=0.9)
custom_optimizer = optim.SGD(custom_model.parameters(), lr
   =0.001, momentum=0.9)
```

9. Funkcja pętli trenowania

Definiujemy funkcję pętli trenowania, która będzie:

1) Trenować model na zbiorze treningowym.

2) Obliczać stratę i dokładność na zbiorze treningowym.

3) Walidować model na zbiorze walidacyjnym.

4) Obliczać stratę i dokładność na zbiorze walidacyjnym.

5) Wyświetlać wyniki dla każdej epoki.

```
# Training loop function
def train_model(model, optimizer, num_epochs=5):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        correct = 0
        total = 0

        for i, (inputs, labels) in enumerate(train_loader):
            optimizer.zero_grad()  # Zero the parameter
   gradients
            outputs = model(inputs)  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Optimize the model

            running_loss += loss.item()
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        train_accuracy = 100 * correct / total
        train_loss = running_loss / len(train_loader)

        model.eval()
        val_loss = 0.0
        val_correct = 0
        val_total = 0

        with torch.no_grad():
            for inputs, labels in val_loader:
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                val_loss += loss.item()
                _, predicted = torch.max(outputs.data, 1)
                val_total += labels.size(0)
                val_correct += (predicted == labels).sum().item
   ()

        val_accuracy = 100 * val_correct / val_total
        val_loss /= len(val_loader)

        print(f"Epoch [{epoch + 1}/{num_epochs}], "
              f"Train Loss: {train_loss:.4f}, Train Accuracy: {
   train_accuracy:.2f}%, "
              f"Validation Loss: {val_loss:.4f}, Validation
   Accuracy: {val_accuracy:.2f}%")
```

10. Trenowanie obu modeli

Teraz przeprowadzimy trening obu modeli: niestandardowego modelu CNN oraz zmodyfikowanego ResNet-18.

```
# Train both models
```

```
print("Training Custom Model:")
train_model(custom_model, custom_optimizer)

print("\nTraining ResNet-18 Model:")
train_model(resnet_model, resnet_optimizer)
```

11. Zapisywanie wytrenowanego modelu

Zapisujemy wytrenowany model ResNet-18, aby można było go używać w przyszłości.

```
# Save the trained model
torch.save(resnet_model.state_dict(), 'caltech101_resnet18.pth')
```

12. Tworzenie interfejsu Gradio do interakcji z modelem

Ładujemy zapisany model ResNet-18, definiujemy funkcję predykcji i tworzymy interfejs Gradio.

12.1. Ładowanie modelu

Ładujemy zapisany model ResNet-18 do stanu ewaluacji.

```
# Load the saved model for inference
model = resnet18(pretrained=False)
model.fc = nn.Linear(model.fc.in_features, num_classes)
model.load_state_dict(torch.load('caltech101_resnet18.pth'))
model.eval()
```

12.2. Definiowanie funkcji predykcji

Definiujemy funkcję predykcji, która przetwarza obraz, przekazuje go przez model i zwraca wyniki klasyfikacji.

```
# Define the prediction function
def predict(im):
    transform = transforms.Compose([
        transforms.Resize((224, 224)),  # Resize images to 224
   x224
        transforms.Grayscale(num_output_channels=3),  # Convert
   grayscale images to 3 channels
        transforms.ToTensor(),  # Convert images to tensor
        transforms.Normalize(mean=[0.485, 0.456, 0.406], std
   =[0.229, 0.224, 0.225])  # Normalize images
    ])
    im = transform(im).unsqueeze(0)  # Transform and add batch
   dimension
    with torch.no_grad():
        output = model(im)  # Forward pass
    output = torch.nn.functional.softmax(output[0], dim=0)  #
   Apply softmax
```

```
    return {train_dataset.dataset.categories[i]: float(output[i
    ]) for i in range(len(output))}
```

12.3. Tworzenie interfejsu Gradio

Tworzymy interfejs Gradio, który umożliwia użytkownikowi przesyłanie obrazów i wyświetla wyniki klasyfikacji.

```
# Create Gradio interface
imagein = gr.inputs.Image(type='pil')
label = gr.outputs.Label(num_top_classes=5)

gr.Interface(fn=predict, inputs=imagein, outputs=label).launch()
```

13. Tenowanie Modelu do Segmentacji Obrazów za Pomocą Pytoch i DeepLabV3

W tym celu wykorzystamy model DeepLabV3 do segmentacji obrazów na zbiorze danych VOCSegmentation orz użyjemy Google Colab do przyspieszenia trenowania modelu.

14. Uruchamianie notebooka na Google Colab

Aby uruchomić poniższy kod na Google Colab, wykonaj następujące kroki:

14.1. Otwieranie linku w Google Colab

Najpierw otwórz przygotowany notebook w Google Colab, klikając na poniższy link: `https://colab.research.google.com/drive/1k-mWbJZAUwagcJjy9Hrz5qbJU_zhwm7n?usp=sharing`

14.2. Zmiana urządzenia wykonawczego na GPU

Aby przyspieszyć trenowanie modeli, zmień urządzenie wykonawcze na GPU:

15. Wprowadzenie

W tej części omówimy kluczowe funkcje i klasy zawarte w kodzie do segmentacji obrazów za pomocą modelu DeepLabV3, używając zbioru danych VOCSegmentation. Instrukcja ma na celu naukę PyTorcha poprzez szczegółowe wyjaśnienia i przykłady.

16. Importowanie bibliotek

Na początku zaimportujemy wszystkie niezbędne biblioteki do przetwarzania obrazów, trenowania modeli, ładowania danych oraz interakcji z modelem za pomocą Gradio.

```
import torch
import torchvision.transforms as transforms
from torch.utils.data import DataLoader, random_split
from torchvision.datasets import VOCSegmentation
from torchvision.models.segmentation import deeplabv3_resnet50
import torch.nn as nn
import torch.optim as optim
import gradio as gr
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
```

17. Definiowanie transformacji i ładowanie danych

Zdefiniujemy niezbędne transformacje i załadujemy zbiór danych VOCSegmentation.

```
# Define transformations for input images and target masks
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])

target_transform = transforms.Compose([
    transforms.Resize((224, 224), interpolation=Image.NEAREST),
    transforms.ToTensor()
])

# Load the VOC Segmentation dataset
dataset = VOCSegmentation(root='./data', year='2012', image_set=
   'train', download=True, transform=transform, target_transform
   =target_transform)

# Split dataset into training and validation sets
train_size = int(0.8 * len(dataset))
val_size = len(dataset) - train_size
train_dataset, val_dataset = random_split(dataset, [train_size,
   val_size])

train_loader = DataLoader(train_dataset, batch_size=8, shuffle=
   True)
val_loader = DataLoader(val_dataset, batch_size=8, shuffle=False
   )
```

18. Ustawianie wstępnie wytrenowanego modelu

Załadujemy wstępnie wytrenowany model DeepLabV3 i dostosujemy go do zadania segmentacji.

```
# Load the pre-trained DeepLabV3 model
model = deeplabv3_resnet50(pretrained=True)

# Modify the final classifier layer to match the number of
   classes in VOCSegmentation
num_classes = 21  # VOCSegmentation has 21 classes including
   background
model.classifier[4] = nn.Conv2d(256, num_classes, kernel_size
   =(1, 1))

# Move the model to the GPU if available
device = torch.device("cuda" if torch.cuda.is_available() else "
   cpu")
model = model.to(device)
```

19. Trenowanie modelu

Zdefiniujemy funkcję straty, optymalizator oraz pętlę trenowania, śledząc zarówno dokładność, jak i stratę podczas trenowania i walidacji.

```
# Define loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum
   =0.9)

# Training loop function
def train_model(model, optimizer, num_epochs=5):
    for epoch in range(num_epochs):
        model.train()
        running_loss = 0.0
        for i, (inputs, labels) in enumerate(train_loader):
            inputs = inputs.to(device)
            labels = labels.squeeze(1).long().to(device)  #
   Ensure labels are single-channel and long type

            optimizer.zero_grad()  # Zero the parameter
   gradients
            outputs = model(inputs)['out']  # Forward pass
            loss = criterion(outputs, labels)  # Compute loss
            loss.backward()  # Backward pass
            optimizer.step()  # Optimize the model

            running_loss += loss.item()

        train_loss = running_loss / len(train_loader)

        model.eval()
        val_loss = 0.0

        with torch.no_grad():
            for inputs, labels in val_loader:
                inputs = inputs.to(device)
                labels = labels.squeeze(1).long().to(device)  #
   Ensure labels are single-channel and long type
                outputs = model(inputs)['out']
                loss = criterion(outputs, labels)
                val_loss += loss.item()

        val_loss /= len(val_loader)

        print(f"Epoch [{epoch + 1}/{num_epochs}], Train Loss: {
   train_loss:.4f}, Validation Loss: {val_loss:.4f}")

# Train the model
train_model(model, optimizer)
```

20. Zapisywanie wytrenowanego modelu

Zapiszemy wytrenowany model, aby można było go później używać.

```
# Save the trained model
torch.save(model.state_dict(), 'deeplabv3_vocsegmentation.pth')
```

21. Ładowanie wytrenowanego modelu i segmentacja obrazu

Załadujemy wytrenowany model, przeprowadzimy segmentację obrazu i wyświetlimy wyniki.

```
# Load the pre-trained DeepLabV3 model
model = deeplabv3_resnet50(pretrained=True)
model.eval()

# Define the device
device = torch.device("cuda" if torch.cuda.is_available() else "
    cpu")
model = model.to(device)

# Define the color map for segmentation classes
VOC_COLORMAP = np.array([
    [0, 0, 0], [128, 0, 0], [0, 128, 0], [128, 128, 0], [0, 0,
    128],
    [128, 0, 128], [0, 128, 128], [128, 128, 128], [64, 0, 0],
    [192, 0, 0],
    [64, 128, 0], [192, 128, 0], [64, 0, 128], [192, 0, 128],
    [64, 128, 128],
    [192, 128, 128], [0, 64, 0], [128, 64, 0], [0, 192, 0],
    [128, 192, 0],
    [0, 64, 128]
])

def decode_segmentation_mask(mask, colormap):
    r = np.zeros_like(mask).astype(np.uint8)
    g = np.zeros_like(mask).astype(np.uint8)
    b = np.zeros_like(mask).astype(np.uint8)

    for l in range(0, len(colormap)):
        idx = mask == l
        r[idx] = colormap[l, 0]
        g[idx] = colormap[l, 1]
        b[idx] = colormap[l, 2]

    rgb = np.stack([r, g, b], axis=2)
    return rgb

# Define the transformation for the input image
transform = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
    0.224, 0.225])
])

# Load and preprocess the image
image_path = "asd.jpg"  # Change this to the path of your image
```

```
image = Image.open(image_path)
input_image = transform(image).unsqueeze(0).to(device)

# Perform the segmentation
with torch.no_grad():
    output = model(input_image)['out']
output = torch.argmax(output.squeeze(), dim=0).cpu().numpy()

# Convert the segmentation mask to an RGB image
segmentation_mask = decode_segmentation_mask(output,
   VOC_COLORMAP)

# Display the original image and the segmentation result
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
ax[0].imshow(image)
ax[0].set_title("Original Image")
ax[0].axis('off')

ax[1].imshow(segmentation_mask)
ax[1].set_title("Segmentation Result")
ax[1].axis('off')

plt.show()
```