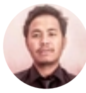


Santosh Premi Adhikari
Posted on Sep 1, 2024

Fine-Tuning a Pre-Trained Model in PyTorch: A Step-by-Step Guide for Beginners

#ai #machinelearning #finetuning #computervision

Fine-tuning is a powerful technique that allows you to adapt a pre-trained model to a new task, saving time and resources. This tutorial will guide you through fine-tuning a ResNet18 model for digit classification using PyTorch.

Step 1: Setting Up the Environment and Model

First, let's import the required libraries and set up the pre-trained model.

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torchvision import models

# Load a pre-trained ResNet18 model
```

```
model = models.resnet18(pretrained=True)

# Modify the last layer to match MNIST classes (10 classe
model.fc = nn.Linear(model.fc.in_features, 10)

# Set the model to training mode and use GPU if availabl
device = torch.device("cuda" if torch.cuda.is_available(
model = model.to(device)
```

Explanation: We are using ResNet18, which is pre-trained on ImageNet. The last layer is modified to fit our new task (10 classes for digits 0-9).

Step 2: Preparing the Dataset

We need to resize the MNIST images to 224x224 because ResNet18 expects this size.

```
# Transform images to 224x224 and normalize
transform = transforms.Compose([
    transforms.Resize(224),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Load the MNIST dataset
trainset = torchvision.datasets.MNIST(root='./data', tra
trainloader = torch.utils.data.DataLoader(trainset, batc

testset = torchvision.datasets.MNIST(root='./data', trai
testloader = torch.utils.data.DataLoader(testset, batch_
```

Explanation: We resize images and normalize them to fit what the *model expects. We also create data loaders for training and testing.*

Step 3: Setting Up the Loss Function and Optimizer

Define the loss function and optimizer. We use Adam, a popular choice for fine-tuning.

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

# Learning rate scheduler to adjust the learning rate
scheduler = optim.lr_scheduler.StepLR(optimizer, step_si
```

Explanation: CrossEntropyLoss is used for classification tasks, and Adam is chosen for optimization. The scheduler helps adjust the learning rate during training.

Step 4: Fine-Tuning the Model

Train the model for a few epochs to fine-tune it on the new task.

```
# Fine-tune the model
num_epochs = 5
#set num_epochs to a smaller number like 1 and use T4 GPU
for epoch in range(num_epochs):
    running_loss = 0.0
    for images, labels in trainloader:
        images, labels = images.to(device), labels.to(dev

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
```

```
            running_loss += loss.item()

        # Step the scheduler after each epoch
        scheduler.step()

        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {runni

    print('Fine-tuning complete!')
```

Explanation: We loop through the dataset, perform forward and backward passes, and update the model's weights. This step adapts the pre-trained model to our specific task.

Step 5: Saving the Fine-Tuned Model
Save the fine-tuned model for later use.

```
# Save the fine-tuned model
torch.save(model.state_dict(), 'finetuned_resnet18_mnist
print('Model saved!')
```

Explanation: We save the model's state dictionary (parameters) to a file. This allows us to load it later without retraining.

Step 6: Evaluating the Model
Check how well the model performs on unseen data.

```
# Set the model to evaluation mode
model.eval()

correct = 0
total = 0
```

```
with torch.no_grad():
    for images, labels in testloader:
        images, labels = images.to(device), labels.to(dev
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f'Accuracy of the fine-tuned model on the test ima
```

Explanation: We set the model to evaluation mode and calculate accuracy on the test set to see how well the model generalizes to new data.

Step 7: Making Predictions
Finally, use the fine-tuned model to make predictions on a single image.

```
# Load the model for inference
model = models.resnet18()
model.fc = nn.Linear(model.fc.in_features, 10)
model.load_state_dict(torch.load('finetuned_resnet18_mni
model.eval()
model = model.to(device)

# Make a prediction on a single image from the test set
test_image, _ = testset[0]  # Get the first image from th
test_image = test_image.unsqueeze(0).to(device)  # Add a

output = model(test_image)
_, predicted = torch.max(output, 1)

print('Predicted label:', predicted.item())
```

Explanation: We load the saved model, set it to evaluation mode, and use it to predict the class of a new image.

Conclusion

Fine-tuning is an efficient way to adapt powerful pre-trained models to new tasks with minimal effort. In this guide, you learned how to:

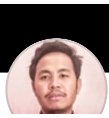
1.Set up a pre-trained model and modify it for a new task.
2.Prepare and load your dataset.
3.Train and fine-tune the model with a new dataset.
4.Save and load the model for future use.
5.Make predictions using the fine-tuned model.

Hope you found this post helpful and enjoyable.
Thank you!

Read More

Top comments (0)

Code of Conduct • Report abuse



Santosh Premi Adhikari

Santosh Premi Adhikari

Fullstack Software Engineer

LOCATION
Bavaria, Germany

EDUCATION
Master Computer Science

WORK
Software Engineer

JOINED
Mar 31, 2023

More from Santosh Premi Adhikari

Type of Neural Networks
#ai #machinelearning #deeplearning #computervision

Building Basic model for Understanding ML
#machinelearning #ai #computervision