Transfer Learning tutorial

Author: Sasank Chilamkurthy

In this tutorial, you will learn how to train your network using transfer learning. You can read more about the transfer learning at cs231n notes

Quoting these notes,

> In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest.

These two major transfer learning scenarios look as follows:

- **Finetuning the convnet**: Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on imagenet 1000 dataset. Rest of the training looks as usual.
- **ConvNet as fixed feature extractor**: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

```
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

import torch
import torch.nn as nn
import torch.optim as optim
import torch.optim.lr_scheduler
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy

plt.ioff()  # interactive mode
```

Load Data

We will use torchvision and torchvision.data packages for loading the data.

The problem we're going to solve today is to train a model to classify **ants** and **bees**. We have about 120 training images each for ants and bees. There are 75 validation images for each class. Usually this is a very small dataset to generalize upon, if trained from scratch. Since we are using transfer learning, we should be able to generalize reasonably well.

This dataset is a very small subset of imagenet.

> **Note**
>
> Download the data from here and extract it to the current directory.

```
# Data augmentation and normalization for training
# Just normalization for validation
data_transforms = {
    'train': transforms.Compose([
        transforms.RandomResizedCrop(224),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ]),
    'val': transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
    ],
}

data_dir = 'hymenoptera_data'
image_datasets = {x: datasets.ImageFolder(os.path.join(data_dir, x),
                                          data_transforms[x])
                  for x in ['train', 'val']}
dataloaders = {x: torch.utils.data.DataLoader(image_datasets[x], batch_size4,
                                              shuffleTrue, num_workers4)
               for x in ['train', 'val']}
dataset_sizes = {x: len(image_datasets[x]) for x in ['train', 'val']}
class_names = image_datasets['train'].classes

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

Visualize a few images

Let's visualize a few training images so as to understand the data augmentations.

```
def imshow(inp, title=None):
    # Imshow for Tensor.
    inp = inp.numpy().transpose((1, 2, 0))
    mean = np.array([0.485, 0.456, 0.406])
    std = np.array([0.229, 0.224, 0.225])
    inp = std * inp + mean
    inp = np.clip(inp, 0, 1)
    plt.imshow(inp)
    if title is None:
        pass
        # pause a bit so that plots are updated
        plt.pause(0.001)

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

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

imshow(out, title=[class_name[x] for x in classes])
```

Training the model

Now, let's write a general function to train a model. Here, we will illustrate:

- Scheduling the learning rate
- Saving the best model

In the following, parameter `scheduler` is an LR scheduler object from `torch.optim.lr_scheduler`.

```
def train_model(model, criterion, optimizer, scheduler, num_epochs25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc0.0

    for epoch in range(num_epochs):
        print('Epoch {}/{}'.format(epoch, num_epochs - 1))
        print('-' * 10)

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

            running_loss = 0.0
            running_corrects = 0

            # Iterate over data.
            for inputs, labels in dataloaders[phase]:
                inputs = inputs.device
                labels = labels.device

                # zero the parameter gradients
                optimizer.zero_grad()

                # forward
                # track history if only in train
                with torch.set_grad_enabled(phase == 'train'):
                    outputs = model(inputs)
                    _, preds = torch.max(outputs, 1)
                    loss = criterion(outputs, labels)

                    # backward + optimize only if in training phase
                    if phase == 'train':
                        loss.backward()
                        optimizer.step()

                # statistics
                running_loss += loss.item() * inputs.size(0)
                running_corrects += torch.sum(preds == labels.data)
            epoch_loss = running_loss / dataset_sizes[phase]
            epoch_acc = running_corrects.double() / dataset_sizes[phase]

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

            # deep copy the model
            if phase == 'val' and epoch_acc > best_acc:
                best_acc = epoch_acc
                best_model_wts = copy.deepcopy(model.state_dict())

        print()

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

    # load best model weights
    model.load_state_dict(best_model_wts)
    return model
```

Visualizing the model predictions

Generic function to display predictions for a few images

```
def visualize_model(model, num_images=6):
    was_training = model.training
    model.eval()
    images_so_far = 0
    fig = plt.figure()

    with torch.no_grad():
        for i, (inputs, labels) in enumerate(dataloaders['val']):
            inputs = inputs.device
            labels = labels.device

            outputs = model(inputs)
            _, preds = torch.max(outputs, 1)

            for j in range(inputs.size()[0]):
                images_so_far += 1
                ax = plt.subplot(num_images//2, 2, images_so_far)
                ax.axis('off')
                ax.title('predicted: {}'.format(class_names[preds[j]]))
                imshow(inputs.cpu()[j])

                if images_so_far == num_images:
                    model.train(was_training)
                    return
    model.train(was_training)
```

Finetuning the convnet

Load a pretrained model and reset final fully connected layer.

```
model_ft = models.resnet18(pretrainedTrue)
num_ftftrs = model_ft.fc.in_features
# Here the size of each output sample in 2
model_ft = nn.Linear(num_ftftrs, 2)

model_ft = model_ft.device()

criterion = nn.CrossEntropyLoss()

# Observe that all parameters are being optimized
optimizer_ft = optim.SGD(model_ft.parameters(), lr0.001, momentum0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_ft, step_size7, gamma0.1)
```

Train and evaluate

It should take around 15-25 min on CPU. On GPU though, it takes less than a minute.

```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs25)
```

Out:

```
Epoch 0/24
----------
train Loss: 0.5998 Acc: 0.7131
val Loss: 0.2062 Acc: 0.9150

Epoch 1/24
----------
train Loss: 0.5239 Acc: 0.7951
val Loss: 0.2104 Acc: 0.9150

Epoch 2/24
----------
train Loss: 0.7201 Acc: 0.7336
val Loss: 0.2911 Acc: 0.8431

Epoch 3/24
----------
train Loss: 0.4708 Acc: 0.8238
val Loss: 0.2092 Acc: 0.8954

Epoch 4/24
----------
train Loss: 0.3891 Acc: 0.8566

...
```

```
visualize_model(model_ft)
```

ConvNet as fixed feature extractor

Here, we need to freeze all the network except the final layer. We need to set `requires_grad == False` to freeze the parameters so that the gradients are not computed in `backward()`.

You can read more about this in the documentation here.

```
model_conv = torchvision.models.resnet18(pretrainedTrue)
for param in model_conv.parameters():
    param.requires_grad = False

# Parameters of newly constructed modules have requires_grad=True by default
num_ftftrs = model_conv.fc.in_features
model_conv.fc = nn.Linear(num_ftftrs, 2)

model_conv = model_conv.device()

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(), lr0.001, momentum0.9)

# Decay LR by a factor of 0.1 every 7 epochs
exp_lr_scheduler = lr_scheduler.StepLR(optimizer_conv, step_size7, gamma0.1)
```

Train and evaluate

On CPU this will take about half the time compared to previous scenario. This is expected as gradients don't need to be computed for most of the network. However, forward does need to be computed.

```
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs25)
```

Out:

```
Epoch 0/24
----------
train Loss: 0.6752 Acc: 0.6230
val Loss: 0.2135 Acc: 0.9214

Epoch 1/24
----------
train Loss: 0.4737 Acc: 0.7810
val Loss: 0.1963 Acc: 0.9412

Epoch 2/24
----------
train Loss: 0.4814 Acc: 0.7705
val Loss: 0.2054 Acc: 0.9085

Epoch 3/24
----------
train Loss: 0.4841 Acc: 0.7624
val Loss: 0.4769 Acc: 0.8170

Epoch 4/24
----------
train Loss: 0.5135 Acc: 0.7705

...
```

```
visualize_model(model_conv)

plt.ioff()
plt.show()
```

Total running time of the script: (1 minutes 33.371 seconds)

Download Python source code: transfer_learning_tutorial.py

Download Jupyter notebook: transfer_learning_tutorial.ipynb

Gallery generated by Sphinx-Gallery

Previous Next

© Copyright 2017, Pytorch.

Built with Sphinx using a theme provided by Read the Docs.

PyTorch Tutorials
PyTorch
DAD
Search docs
BEGINNER TUTORIALS
Deep Learning with PyTorch: A 60 Minute Blitz
PyTorch for former Torch users
Learning PyTorch with Examples
Transfer Learning tutorial
Quick Start
Training the model
Finetuning the convnet
ConvNet as fixed feature extractor
Data Loading and Processing Tutorial
Deep Learning for NLP with Pytorch
INTERMEDIATE TUTORIALS
Classifying Names with a Character-Level RNN
Generating Names with a Character-Level RNN
Translation with a Sequence to Sequence Network and Attention
Reinforcement Learning (DQN) tutorial
Writing Distributed Applications with PyTorch
Spatial Transformer Networks Tutorial
ADVANCED TUTORIALS
Neural Transfer with PyTorch
Creating extensions using numpy and scipy