transfer_learning_tutorial

February 18, 2022

[1]:
```
%matplotlib inline
```

1 Transfer Learning for Computer Vision Tutorial

Author: Sasank Chilamkurthy <https://chsasank.github.io>_

In this tutorial, you will learn how to train a convolutional neural network for image classification using transfer learning. You can read more about the transfer learning at cs231n notes <https://cs231n.github.io/transfer-learning/>___

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.

[2]:
```
# License: BSD
# Author: Sasank Chilamkurthy

from __future__ import print_function, division

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
import copy

cudnn.benchmark = True
plt.ion()   # interactive mode
```

[2]: <matplotlib.pyplot._IonContext at 0x7fb0b8ea76a0>

1.1 Load Data

We will use torchvision and torch.utils.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 <https://download.pytorch.org/tutorial/hymenoptera_data.zip>_` and extract it to the current directory.

[3]:
```
# 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 = '../source/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_size=4,
```

```
                                              shuffle=True, num_workers=4)
              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")
```

```
/home/runner/.local/lib/python3.8/site-
packages/torch/utils/data/dataloader.py:478: UserWarning: This DataLoader will
create 4 worker processes in total. Our suggested max number of worker in
current system is 2, which is smaller than what this DataLoader is going to
create. Please be aware that excessive worker creation might get DataLoader
running slow or even freeze, lower the worker number to avoid potential
slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

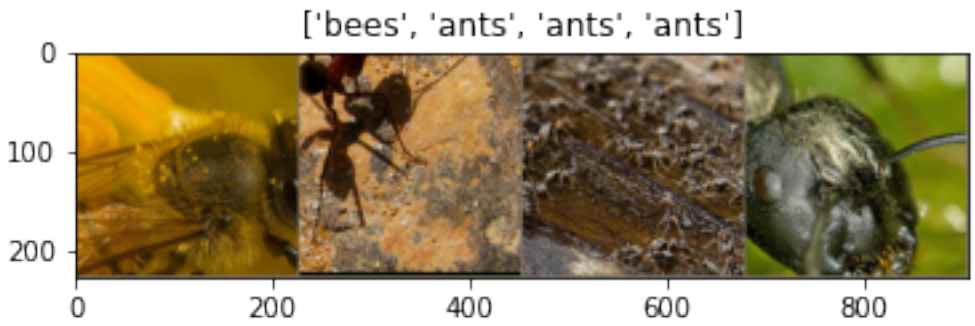
Visualize a few images ^^^^^^^^^^^^^^^^^^^^^^^^ Let's visualize a few training images so as to understand the data augmentations.

[4]:
```
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 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 batch
out = torchvision.utils.make_grid(inputs)

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



1.2 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`.

[5]:
```
def train_model(model, criterion, optimizer, scheduler, num_epochs=25):
    since = time.time()

    best_model_wts = copy.deepcopy(model.state_dict())
    best_acc = 0.0

    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':
                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.to(device)
                labels = labels.to(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)
            if phase == 'train':
                scheduler.step()

            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

[6]:
```
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.to(device)
            labels = labels.to(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.set_title('predicted: {}'.format(class_names[preds[j]]))
                imshow(inputs.cpu().data[j])

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

1.3 Finetuning the convnet

Load a pretrained model and reset final fully connected layer.

[7]:
```
model_ft = models.resnet18(pretrained=True)
num_ftrs = model_ft.fc.in_features
# Here the size of each output sample is set to 2.
# Alternatively, it can be generalized to nn.Linear(num_ftrs, len(class_names)).
model_ft.fc = nn.Linear(num_ftrs, 2)

model_ft = model_ft.to(device)

criterion = nn.CrossEntropyLoss()

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

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

```
Downloading: "https://download.pytorch.org/models/resnet18-f37072fd.pth" to
/home/runner/.cache/torch/hub/checkpoints/resnet18-f37072fd.pth
```

```
  0%|          | 0.00/44.7M [00:00<?, ?B/s]
```

Train and evaluate ^^^^^^^^^^^^^^^^^^^

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

[8]:
```
model_ft = train_model(model_ft, criterion, optimizer_ft, exp_lr_scheduler,
                       num_epochs=25)
```

```
Epoch 0/24
----------
train Loss: 0.6558 Acc: 0.7090
val Loss: 0.2453 Acc: 0.9150

Epoch 1/24
----------
train Loss: 0.8253 Acc: 0.7049
val Loss: 0.4293 Acc: 0.8497

Epoch 2/24
----------
train Loss: 0.4105 Acc: 0.8443
val Loss: 0.2353 Acc: 0.9216

Epoch 3/24
----------
train Loss: 0.5289 Acc: 0.8033
val Loss: 0.2572 Acc: 0.8889

Epoch 4/24
----------
train Loss: 0.5840 Acc: 0.7623
val Loss: 0.2544 Acc: 0.9020

Epoch 5/24
----------
train Loss: 0.4483 Acc: 0.8156
val Loss: 0.3800 Acc: 0.8693

Epoch 6/24
----------
train Loss: 0.5342 Acc: 0.8238
val Loss: 0.2795 Acc: 0.8954

Epoch 7/24
----------
train Loss: 0.4526 Acc: 0.8443
val Loss: 0.1786 Acc: 0.9412
```

```
Epoch 8/24
----------
train Loss: 0.4197 Acc: 0.8443
val Loss: 0.1831 Acc: 0.9346

Epoch 9/24
----------
train Loss: 0.2762 Acc: 0.8934
val Loss: 0.1695 Acc: 0.9346

Epoch 10/24
----------
train Loss: 0.3834 Acc: 0.8402
val Loss: 0.1933 Acc: 0.9216

Epoch 11/24
----------
train Loss: 0.4257 Acc: 0.8197
val Loss: 0.1890 Acc: 0.9216

Epoch 12/24
----------
train Loss: 0.3064 Acc: 0.8648
val Loss: 0.1824 Acc: 0.9346

Epoch 13/24
----------
train Loss: 0.3091 Acc: 0.8730
val Loss: 0.1855 Acc: 0.9346

Epoch 14/24
----------
train Loss: 0.2423 Acc: 0.8893
val Loss: 0.1892 Acc: 0.9281

Epoch 15/24
----------
train Loss: 0.3483 Acc: 0.8648
val Loss: 0.1974 Acc: 0.9216

Epoch 16/24
----------
train Loss: 0.2354 Acc: 0.8893
val Loss: 0.2264 Acc: 0.9085

Epoch 17/24
----------
train Loss: 0.2829 Acc: 0.8893
```

```
val Loss: 0.1821 Acc: 0.9346

Epoch 18/24
----------
train Loss: 0.2360 Acc: 0.8852
val Loss: 0.1865 Acc: 0.9346

Epoch 19/24
----------
train Loss: 0.2870 Acc: 0.8852
val Loss: 0.2156 Acc: 0.8954

Epoch 20/24
----------
train Loss: 0.2580 Acc: 0.9016
val Loss: 0.1964 Acc: 0.9346

Epoch 21/24
----------
train Loss: 0.2754 Acc: 0.8770
val Loss: 0.1919 Acc: 0.9346

Epoch 22/24
----------
train Loss: 0.2241 Acc: 0.9139
val Loss: 0.1985 Acc: 0.9346

Epoch 23/24
----------
train Loss: 0.2465 Acc: 0.9098
val Loss: 0.1774 Acc: 0.9412

Epoch 24/24
----------
train Loss: 0.2642 Acc: 0.8648
val Loss: 0.1874 Acc: 0.9281

Training complete in 9m 23s
Best val Acc: 0.941176
```

[9]:
```
visualize_model(model_ft)
```

predicted: bees

predicted: bees

predicted: bees

predicted: bees

predicted: bees

1.4 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 <https://pytorch.org/docs/notes/autograd.html#excluding-subgraphs-from-backward>__`.

[10]:
```
model_conv = torchvision.models.resnet18(pretrained=True)
for param in model_conv.parameters():
    param.requires_grad = False

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

model_conv = model_conv.to(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(), lr=0.001, momentum=0.9)

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

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.

[11]:
```
model_conv = train_model(model_conv, criterion, optimizer_conv,
                         exp_lr_scheduler, num_epochs=25)
```

```
Epoch 0/24
----------
train Loss: 0.6195 Acc: 0.6803
val Loss: 0.2196 Acc: 0.9346

Epoch 1/24
----------
train Loss: 0.5184 Acc: 0.7254
val Loss: 0.2432 Acc: 0.8889

Epoch 2/24
----------
train Loss: 0.3808 Acc: 0.8156
val Loss: 0.1876 Acc: 0.9477

Epoch 3/24
----------
train Loss: 0.4313 Acc: 0.8279
val Loss: 0.1976 Acc: 0.9412

Epoch 4/24
----------
train Loss: 0.5221 Acc: 0.7500
val Loss: 0.1754 Acc: 0.9542

Epoch 5/24
----------
train Loss: 0.3392 Acc: 0.8648
val Loss: 0.1963 Acc: 0.9346

Epoch 6/24
----------
train Loss: 0.3882 Acc: 0.8484
val Loss: 0.2218 Acc: 0.9281

Epoch 7/24
----------
train Loss: 0.4173 Acc: 0.7992
val Loss: 0.1955 Acc: 0.9412

Epoch 8/24
----------
train Loss: 0.3098 Acc: 0.8648
val Loss: 0.2167 Acc: 0.9412

Epoch 9/24
----------
train Loss: 0.3127 Acc: 0.8607
```

```
val Loss: 0.1951 Acc: 0.9346

Epoch 10/24
----------
train Loss: 0.3342 Acc: 0.8607
val Loss: 0.1991 Acc: 0.9346

Epoch 11/24
----------
train Loss: 0.3375 Acc: 0.8607
val Loss: 0.1958 Acc: 0.9346

Epoch 12/24
----------
train Loss: 0.3857 Acc: 0.7992
val Loss: 0.2175 Acc: 0.9281

Epoch 13/24
----------
train Loss: 0.3758 Acc: 0.8320
val Loss: 0.2001 Acc: 0.9412

Epoch 14/24
----------
train Loss: 0.3281 Acc: 0.8730
val Loss: 0.2103 Acc: 0.9412

Epoch 15/24
----------
train Loss: 0.3686 Acc: 0.8115
val Loss: 0.1980 Acc: 0.9477

Epoch 16/24
----------
train Loss: 0.3100 Acc: 0.8811
val Loss: 0.1990 Acc: 0.9477

Epoch 17/24
----------
train Loss: 0.3168 Acc: 0.8566
val Loss: 0.1958 Acc: 0.9412

Epoch 18/24
----------
train Loss: 0.3398 Acc: 0.8402
val Loss: 0.2079 Acc: 0.9412

Epoch 19/24
```

```
----------
train Loss: 0.3216 Acc: 0.8484
val Loss: 0.1903 Acc: 0.9412

Epoch 20/24
----------
train Loss: 0.3687 Acc: 0.8238
val Loss: 0.1931 Acc: 0.9346

Epoch 21/24
----------
train Loss: 0.2530 Acc: 0.9098
val Loss: 0.2072 Acc: 0.9412

Epoch 22/24
----------
train Loss: 0.3766 Acc: 0.8320
val Loss: 0.1978 Acc: 0.9412

Epoch 23/24
----------
train Loss: 0.3159 Acc: 0.8730
val Loss: 0.1950 Acc: 0.9412

Epoch 24/24
----------
train Loss: 0.2991 Acc: 0.8893
val Loss: 0.2291 Acc: 0.9412

Training complete in 4m 42s
Best val Acc: 0.954248
```

[12]:
```
visualize_model(model_conv)

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

predicted: ants

predicted: bees

predicted: bees

predicted: ants

predicted: ants

1.5 Further Learning

If you would like to learn more about the applications of transfer learning, checkout our `Quantized Transfer Learning for Computer Vision Tutorial <https://pytorch.org/tutorials/intermediate/quantized_transfer_learning_tutorial.html>_`.