Open in Colab

Transfer Learning with Pytorch

These have been adapted from Transfer Learning for Computer Vision Tutorial.

In this notebook we'll be using the Caltech 256 dataset (which we've used for a bunch of the examples in this module) to show how we can adapt a pretrained model to a new dataset.

All of the pretrained models were trained on the ImageNet dataset. Specifically, these networks were trained to recognize 1,000 different object classes.

Transfer learning is a branch of machine learning that focuses on using knowledged acquired from one learning problem and applying it to another. In the case of a neural network, what we'll wind up doing is modifying the network trained on ImageNet so that it can be used on the Caltech data. Specifically, we'll be chopping off the last layer of the pretrained network and replacing it with a linear layer that produces the 257 numbers (there are 256 object classes in Caltech 256 and a "clutter" category) necessary for prediction in the Caltech 256 dataset.

Once we've modified our network, we can train it on the Caltech 256 Dataset. The important caveat here is that we are going to freeze the weights for all but our newly added final layer. This will allow the network to fairly rapidly converge to a decent solution. You can also modify the code so that all weights in the network are trained and compare the results.

In [0]:

```
# standard importants for pytorch and torchvision
from __future__ import print_function
from __future__ import division
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import torchvision
from torchvision import datasets, models, transforms
import matplotlib.pyplot as plt
import time
import os
import copy
```

In [0]:

```
# We have created a normalized, precropped, and resized version of Caltech 256
# The images are 224 by 224, which is the standard input size for models trained
# on ImageNet
import gdown
gdown.download('https://drive.google.com/uc?authuser=0&id=16ieLmAEV6Xn4Iw0GwYGQJkeXenEaKcz-
&export=download',
               'caltech_224_224.hdf5',
               quiet=False)
```

In [0]:

```
import h5py
from torch.utils.data import Dataset
from torchvision import transforms

# This is the class that we gave for wrapping an hdf5 file.  It is much faster
# than loading the images from JPGs every time.
class H5Dataset(Dataset):
    def __init__(self, h5_path):
        super(H5Dataset, self).__init__()
        self.h5_file = h5py.File(h5_path, 'r', libver='latest', swmr=True)
        self.target_cache = []
        for i in range(len(self)):
            self.target_cache.append(self.h5_file['targets'][i])

    def __getitem__(self, index):
        return torch.FloatTensor(self.h5_file['data'][index]), self.target_cache[index]
```

```
    def __len__(self):
        return len(self.h5_file['data'])

    def close_dataset(self):
        self.h5_file.close()

cal_tech = H5Dataset('caltech_224_224.hdf5')
```

In [0]:

```
im, target = cal_tech[500]
plt.imshow(im.transpose(0,1).transpose(1,2))
plt.axis('off')
plt.show()
```

In [0]:

```
# Models to choose from [resnet, alexnet, vgg, squeezenet, densenet, inception]
model_name = "resnet"

# Number of classes in the dataset
num_classes = 257

# Batch size for training (change depending on how much memory you have)
batch_size = 32

# Number of epochs to train for
num_epochs = 15

# Flag for feature extracting. When False, we finetune the whole model,
#   when True we only update the reshaped layer params
feature_extract = False
```

In [0]:

```
def train_model(model, dataloaders, criterion, optimizer, num_epochs=25, is_inception=False):
    since = time.time()

    val_acc_history = []

    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)
        batch_num = 0

        # 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]:
                print(phase, "processing batch", batch_num)
                batch_num += 1
                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'):
                    # Get model outputs and calculate loss
                    # Special case for inception because in training it has an auxiliary output. I
train
```

```
train
                    #   mode we calculate the loss by summing the final output and the auxiliary ou
tput
                    #   but in testing we only consider the final output.
                    if is_inception and phase == 'train':
                        # From https://discuss.pytorch.org/t/how-to-optimize-inception-model-with-a
uxiliary-classifiers/7958
                        outputs, aux_outputs = model(inputs)
                        loss1 = criterion(outputs, labels)
                        loss2 = criterion(aux_outputs, labels)
                        loss = loss1 + 0.4*loss2
                    else:
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)

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

                    # 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 / len(dataloaders[phase].dataset)
            epoch_acc = running_corrects.double() / len(dataloaders[phase].dataset)

            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())
            if phase == 'val':
                val_acc_history.append(epoch_acc)

        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, val_acc_history
```

In [0]:

```
# this function freezes the weights of the network, unless we are not in
# feature extracting mode.  The last layer (the one we will train) is added
# after this function is called (so its weights will not be frozen)
def set_parameter_requires_grad(model, feature_extracting):
    if feature_extracting:
        for param in model.parameters():
            param.requires_grad = False
```

In [0]:

```
def initialize_model(model_name, num_classes, feature_extract, use_pretrained=True):
    # Initialize these variables which will be set in this if statement. Each of these
    #   variables is model specific.
    model_ft = None
    input_size = 0

    if model_name == "resnet":
        """ Resnet18
        """
        model_ft = models.resnet18(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs, num_classes)
        # try a more complex final layer by uncommenting the following line
        # I (Paul) didn't adapt this to other network types, but you could
        #model_ft.fc = nn.Sequential(
```

```
        #model_ft.fc = nn.Sequential(
        #        nn.Linear(num_ftrs, 512),
        #        nn.ReLU(),
        #        nn.Dropout(0.4),
        #        nn.Linear(512, 257)
        #)
        input_size = 224

    elif model_name == "alexnet":
        """ Alexnet
        """
        model_ft = models.alexnet(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "vgg":
        """ VGG11_bn
        """
        model_ft = models.vgg11_bn(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier[6].in_features
        model_ft.classifier[6] = nn.Linear(num_ftrs,num_classes)
        input_size = 224

    elif model_name == "squeezenet":
        """ Squeezenet
        """
        model_ft = models.squeezenet1_0(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        model_ft.classifier[1] = nn.Conv2d(512, num_classes, kernel_size=(1,1), stride=(1,1))
        model_ft.num_classes = num_classes
        input_size = 224

    elif model_name == "densenet":
        """ Densenet
        """
        model_ft = models.densenet121(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        num_ftrs = model_ft.classifier.in_features
        model_ft.classifier = nn.Linear(num_ftrs, num_classes)
        input_size = 224

    elif model_name == "inception":
        """ Inception v3
        Be careful, expects (299,299) sized images and has auxiliary output
        """
        model_ft = models.inception_v3(pretrained=use_pretrained)
        set_parameter_requires_grad(model_ft, feature_extract)
        # Handle the auxilary net
        num_ftrs = model_ft.AuxLogits.fc.in_features
        model_ft.AuxLogits.fc = nn.Linear(num_ftrs, num_classes)
        # Handle the primary net
        num_ftrs = model_ft.fc.in_features
        model_ft.fc = nn.Linear(num_ftrs,num_classes)
        input_size = 299

    else:
        print("Invalid model name, exiting...")
        exit()

    return model_ft, input_size

# Initialize the model for this run
model_ft, input_size = initialize_model(model_name, num_classes, feature_extract, use_pretrained=True)

# Print the model we just instantiated
print(model_ft)
```

In [0]:

```
from torch.utils.data.sampler import SubsetRandomSampler

# use 20,000 randomly selected images for training and the rest (~10,000) for testing
n_train = 20000
```

```
n_train = 20000
indices = torch.randperm(len(cal_tech))

train_idx, test_idx = indices[:n_train], indices[n_train:]
train_sampler = SubsetRandomSampler(train_idx)
test_sampler = SubsetRandomSampler(test_idx)

# Get our data into the mini batch size that we defined
train_loader = torch.utils.data.DataLoader(cal_tech, batch_size=batch_size,
                                           sampler=train_sampler, num_workers=2)


test_loader = torch.utils.data.DataLoader(cal_tech, batch_size=batch_size,
                                          sampler=test_sampler, num_workers=2)
```

In [0]:

```
device = 'cuda'
# Send the model to GPU
model_ft = model_ft.to(device)

# Gather the parameters to be optimized/updated in this run. If we are
#  finetuning we will be updating all parameters. However, if we are
#  doing feature extract method, we will only update the parameters
#  that we have just initialized, i.e. the parameters with requires_grad
#  is True.
params_to_update = model_ft.parameters()
print("Params to learn:")
if feature_extract:
    params_to_update = []
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            params_to_update.append(param)
            print("\t",name)
else:
    for name,param in model_ft.named_parameters():
        if param.requires_grad == True:
            print("\t",name)

optimizer_ft = optim.Adam(params_to_update, lr=0.001)
```

In [0]:

```
# Setup the loss fxn
criterion = nn.CrossEntropyLoss()
dataloaders_dict = {'train': train_loader, 'val': test_loader}

# Train and evaluate
model_ft, hist = train_model(model_ft, dataloaders_dict, criterion, optimizer_ft, num_epochs=num_ep
ochs, is_inception=(model_name=="inception"))
```

In [0]:

```
!nvidia-smi
```

In [0]: