Project 2: Transfer Learning in PyTorch

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER, AND ENERGY ENGINEERING,
EEE508: Image and Video Processing and Compression

Adapted from Deep Learning Course Labs by Samuel Dodge and Lina J Karam
©2017-2019. Do not distribute outside this class and do not post.

Use and Distribution of Code Not Allowed
Sharing solutions or partial solutions to the programming assignments is not allowed, whether you do so directly or indirectly. Providing code and receiving code are violations of the academic integrity policy. You are responsible for protecting your code and making sure that no one gets a copy of your code. You should protect accounts and computers that contain your code and make sure no one accesses them whether with your knowledge or not. If you have difficulty with your code, you should seek help from the TA and not show your code to anyone else.
Under no circumstances you should post any code you develop for this project in a publicly available online repository (including but no limited to GitHub for example).

Objectives
In this project, students learn how to use and work with PyTorch and how to use deep learning libraries for computer vision with a focus on image classification using Convolutional Neural Networks and transfer learning.

1 PyTorch Basics
PyTorch [1] is an open source machine learning library that is particularly useful for deep learning. PyTorch contains auto-differentiation, meaning that if we write code using PyTorch functions, we can obtain the derivatives without any additional derivation or code. This saves us from having to implement any backward propagation functions for deep networks. Secondly, PyTorch comes with many functions and classes for common deep learning layers, optimizers, and loss functions. Lastly, PyTorch is able to efficiently run computations on either the CPU or GPU.

1.1 Installation
To install PyTorch run the following commands in the Linux terminal:

```
pip install https://download.pytorch.org/whee/cpu/torch-1.0.1.post2-cp27-cp27mu-linux_x86_64.whl
pip install torchvision
```
The first command installs the basic PyTorch package, and the second installs torchvision which includes functions, classes, and models useful for deep learning for computer vision.

1.2 Tutorial

To start we ask you to complete the following tutorial: http://pytorch.org/tutorials/beginner/pytorch_with_examples.html

2 CIFAR100 Example in PyTorch

Next, we will implement a simple neural network using PyTorch. The code for this example is in the included cifar_pytorch.py file.

PyTorch has a module called nn that contains implementations of the most common layers used for neural networks.

In this example, we will implement our model as a class with forward, __init__, fit and predict functions. The initialization function simply sets up our layers using the layer types in the nn package. In the forward pass we pass the data through our layers and return the output. To implement a ReLU layer, we can use the F.relu function in our forward pass. To implement a “flatten” layer, we can use PyTorch’s view to reshape the data.

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv2d args: (number of input channels (aka as input depth),
        # number of output channels (aka output depth), filter size, stride
        # (optional), padding (optional))
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        # Linear args: (# of entries in flattened input, # of output values
        # corresponding to number of fully connected neurons)
        self.fc1 = nn.Linear(1152, 5)
        # MaxPool2d args: (kernel size, stride)
        self.pool = nn.MaxPool2d(2, 2)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 1152) # -1 here indicates that the system
        # automatically computes the number of rows and only the number of
        # columns are specified by the user
        x = self.fc1(x)
        return x
```

Fit function  Next we will implement a fit function as a class method for training the network. It is also common to see the code for training be implemented outside of the model class in a separate function. To implement a fit function, we need to first define an optimization criterion, and select an optimizer. Next we loop for a number of epochs. In each epoch, we use PyTorch’s DataLoader class to loop through the data in batches. The DataLoader automatically takes care of splitting the data into batches. We access the batches with a simple for loop through the DataLoader. For each
batch, we reset to zero the previously calculated gradients using the optimizers `zero_grad` method. Then we call the forward function, compute the loss, call the backwards function, and perform one optimization step.

```python
def fit(self, trainloader):
    # switch to train mode
    self.train()

    # define loss function
criterion = nn.CrossEntropyLoss()

    # setup SGD
    optimizer = optim.SGD(self.parameters(), lr=0.1, momentum=0.0)

    for epoch in range(20):  # loop over the dataset multiple times
        running_loss = 0.0
        for i, data in enumerate(trainloader, 0):
            # get the inputs
            inputs, labels = data

            # zero the parameter gradients
            optimizer.zero_grad()

            # compute forward pass
            outputs = self.forward(inputs)

            # get loss function
            loss = criterion(outputs, labels)

            # do backward pass
            loss.backward()

            # do one gradient step
            optimizer.step()

            # print statistics
            running_loss += loss.item()

        print('Epoch: %d  loss: %.3f' % (epoch + 1, running_loss / (i+1))
```

Finally it is also useful to provide a predict function to run our model on some test data. This predict function will also use the PyTorch loader.
def predict(self, testloader):
    # switch to evaluate mode
    self.eval()

correct = 0
total = 0
all_predicted = []
with torch.no_grad():
    for images, labels in testloader:
        outputs = self.forward(Variable(images))
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
correct += (predicted == labels).sum().item()
        all_predicted += predicted.numpy().tolist()

    print('Accuracy on test images: %d%% (%d / %d)')
    return all_predicted

Notice that we use torch.nn.module.train() for training and torch.nn.module.eval() for prediction. Function train() sets the module in training mode and function eval() sets the module in evaluation mode. These two functions have an effect only on certain modules. They control certain layers like Dropout and BatchNorm to be developed during training and disabled during evaluation. For simple neural network structures for which we do not have any layers that will be affected by self.train() or self.eval(), it is to delete these. But it is a good habit to include these when implementing a neural network.

We also use torch.no_grad() when doing prediction. This function was introduced starting with PyTorch version 0.4. It disables the gradient calculation. You can use it when you are sure that you will not call the backward function. This will reduce memory consumption and computations consumption.

To evaluate the two models we look at both the final classification accuracy as well as the confusion matrix. The rows of the confusion matrix show the predicted class and the columns show the actual class. This lets us analyze the patterns of misclassifications. The included util.py includes the function plot_confusion_matrix which will plot this matrix. Figure 1 shows the confusion matrix for this CIFAR100 example.

## 3 Transfer Learning

Transfer learning is when we use a model trained on one set of data, and adapt it to another set of data. For image datasets, transfer learning works because many features (e.g., edges) are useful across different image datasets. Transfer learning using neural networks trained on large image datasets is highly successful, and can easily be competitive with other approaches that do not use deep learning. Transfer learning also does not require a huge amount of data, since the pre-trained initialization is a good starting point.

In this project, we will consider a fine-tuning based method that updates the parameters of a pre-trained network (trained initially on another, typically larger, dataset) by retraining the network on the new dataset of interest while starting with the network parameters values that were obtained from pre-training. We will be using networks that have been pre-trained on the ImageNet dataset [2], and adapt them for different datasets.
3.1 Datasets

We have collected four datasets for use in this project (Table 1). You will be assigned one of these datasets to work on. The datasets are subsets of existing datasets. Figure 2 shows example images from these datasets. You will not get credit if you download and submit a model trained fully on these datasets. We want to fine-tune models that were originally trained on the ImageNet dataset. With transfer learning we alleviate some of the problems with using small datasets. Typically if we tried to train a network from scratch on a small dataset, we might experience overfitting problems. But in transfer learning, we start with some network trained on a much larger dataset. Because of this, the features from the pre-trained network are not likely to overfit our data, yet still likely to be useful for classification.

Table 1: Datasets for Transfer Learning in PyTorch

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Description</th>
<th># Categories</th>
<th># Train / # Test</th>
</tr>
</thead>
</table>

3.2 Base Networks

Table 2 shows the base networks that we will consider in this project. Each group will consider one of the networks (see Section 3.5 for group assignments). All of these networks have PyTorch versions trained on the ImageNet dataset [2], but the architectures of the networks vary. The input size and the last layer input size also vary among the networks.

As an example we will be using DenseNet [7] to explain how to do fine-tuning in PyTorch. DenseNet is not assigned to any teams in the project, but the principles for using DenseNet are the same as for using other models. In PyTorch we can load a pre-trained DenseNet model with the command:
Figure 2: Example images from datasets for transfer learning in PyTorch.
import torchvision.models
model = torchvision.models.densenet121(pretrained=True)

It is important to use the pretrained=True argument. Otherwise, the model will be initialized with random weights.

Table 2: Base Networks for Transfer Learning in PyTorch

<table>
<thead>
<tr>
<th>Model</th>
<th>Year</th>
<th>Input Size</th>
<th>Last layer input size</th>
<th>PyTorch model</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet [8]</td>
<td>2012</td>
<td>224 x 224</td>
<td>4096</td>
<td>torchvision.models.alexnet</td>
</tr>
<tr>
<td>VGG16 [9]</td>
<td>2014</td>
<td>224 x 224</td>
<td>4096</td>
<td>torchvision.models.vgg16</td>
</tr>
<tr>
<td>DenseNet121 [7]</td>
<td>2017</td>
<td>224 x 224</td>
<td>1024</td>
<td>torchvision.models.densenet121</td>
</tr>
</tbody>
</table>

3.3 Pre-processing

It is important that we pre-process the images before sending them to the network. Most DNNs preprocess the images to be zero mean and unit standard deviation before training. During testing if we pass an image that is also not normalized (with respect to the training data), then the network output won’t be useful. We can say that the network “expects” the input to be normalized. PyTorch includes a transform module that implements the common transformations, including normalization, used in pre-processing:

import torchvision.transforms as transforms

For normalization we can utilize the built in PyTorch function Normalize. The values used for normalization can be computed from the images in the ImageNet dataset. For each channel in the image there is a separate mean and standard deviation used for normalization.

normalize = transforms.Normalize(mean=[0.485, 0.456, 0.406],
                               std=[0.229, 0.224, 0.225])

To apply our data with our pretrained neural network, we must resize the images to the input size expected by the network. For Alexnet, VGG16, ResNet18, and DenseNet this is 224 x 224 pixels, but for Inception this is 299 x 299 pixels. There are different ways to ensure our input is the correct size. The simplest way is to resize our input to the correct size. This has the disadvantage of potentially stretching the image if there is an aspect ratio mismatch. An alternative method is to resize the image so that the minimum side length is equal to the required size, and then crop out the extra part of the image to get the desired size. For this tutorial, we use the first method with PyTorch’s Resize method:

resize = transforms.Resize((224, 224))

In preprocessing we would like to apply these transformations in a pipeline to every image. PyTorch includes a useful function called Compose to combine the transformations into a single object representing the pipeline:
preprocessor = transforms.Compose([
    resize,
    transforms.ToTensor(),
    normalize,
])

Note the presence of ToTensor() which is needed to convert from the image representation to the PyTorch tensor representation. We can easily add other transformations to this preprocessor object to do more preprocessing, or to add data augmentation.

3.4 Layers as Feature Extractors

The convolutional layers of the base network act each as a feature extractor. This means that we simply run the images through the pre-trained base network, and take outputs from layer(s) as a feature representation of the image. These features are usually good for classification with a shallow machine learning/classification algorithm (e.g., Support Vector Machine or SVM).

To extract features we need to stop the forward pass after a certain layer. As of writing this tutorial, PyTorch doesn’t offer a simple call to extract a certain layer. However it is very easy to change the model to give the output of a certain layer. How exactly to extract a certain layer depends on the implementation of the model. We recommend you look at the source code for the networks available at https://github.com/pytorch/vision/tree/master/torchvision/models.

The PyTorch AlexNet and VGG models are split into a feature extractor stage, and a classifier stage. The feature extractor consists of the convolutional layers, and the classifier consists of the fully connected layers. As an example, we want to extract the output of the layer before the last fully connected layer. The easiest way to do this is to modify the sequential part of the model to remove the second to last layer.

new_classifier = nn.Sequential(*list(model.classifier.children())[:−1])
model.classifier = new_classifier

If we are using a model that does not use the feature extractor/classifier decomposition, we need to modify the forward pass of the model to only compute up to the requested layer. For example, for DenseNet, we can comment out the last layer to extract the features before the final fully connected layer.

def forward(self, x):
    features = self.features(x)
    out = F.relu(features, inplace=True)
    # average pool features and reshape to (batch size, feature size)
    out = F.avg_pool2d(out, kernel_size=7, stride=1).view(features.size(0), −1)
    # out = self.classifier(out) # commented out to get the features instead
    return out

In this part of this project, you will be using the pre-trained base network as a feature extractor and you will need to replace only the last fully connected layer with a suitable fully connected layer that outputs a vector whose size should match the number of categories assigned to you from the
new dataset. Furthermore, you will need to train the parameters of the new fully connected layers using training samples from your assigned dataset categories.

Next we need some way to load the data from our dataset. Again PyTorch provides some convenient tools to do this. We will use the `datasets.ImageFolder` class to load our dataset. The ImageFolder loader assumes a file structure of our data where each of the classes is stored in a separate folder. Next we will use the `torch.utils.data.DataLoader` class to create a loader that can be used to loop through our data with some batch size. We would need to have separate loaders for the training data and the testing data. A loader can be constructed with:

```python
loader = torch.utils.data.DataLoader(
    datasets.ImageFolder(data_dir, preprocessor),
    batch_size=batch_size,
    shuffle=True)
```

With the loader set, we can now loop through the data and extract features. During looping we can use Python's `enumerate` function to keep track of the batch index. The loader returns a tuple with the data and the target label.

In the case of testing, we don’t need to feed the label to the network, but it would be useful to save the label so that we can use it later for computing the classification accuracy for the test set. To use the input data in our PyTorch model, we need to wrap it as a PyTorch `Variable`.

```python
for i, (in_data, target) in enumerate(loader):
    input_var = torch.autograd.Variable(in_data, volatile=True)
    output = model(input_var)
```

We can assess the classification performance of the model using the classification accuracy on the entire testing set. In addition to this, we can plot a confusion matrix, which tells more information about the errors that the model made. `util.py` includes the function `plot_confus_matrix` which will plot this matrix.

### Important:

The `plot_confus_matrix` function plots a confusion matrix. Note that there is an input parameter `size` to indicate how many classes you want to show in the plot. The default value for `size` is `None` in which case it will plot the whole confusion matrix. If `size` is given some integer value, for example 9, it will plot the confusion matrix from class 0 to class 8. You will need to plot the confusion matrix for the dataset categories that are assigned to your team.

### 3.5 Fine-tuning

The above feature extractor approach, which makes use of the pre-trained base network and only replaces and train the last fully connected layer, might give decent classification results. This is because, for certain problems, the intermediate representations learned by the pre-trained network can be very useful for the new problem. However, even if the pre-trained filters are giving good performance, we may be able to achieve even greater performance by allowing all the parameters of the pre-trained model to adapt to our new dataset. This adaptation process is called fine-tuning.

Performing fine-tuning is exactly the same process as performing training. Refer to the included `cifar_pytorch.py` to see how to train in PyTorch. The main differences in this case is that we want to start from the pre-trained model. We can use the same `DataLoader` and `transform` modules that
we used for feature extraction. During fine-tuning we will usually use a very small learning rate, except for the last fully connected layer which will need to be replaced by a new fully connected layer whose output should match in size the number of our assigned dataset categories. This new fully connected layer needs to be trained from scratch using a relatively larger learning rate than the one used for training the pre-trained layers. For these pre-trained layers, we want to adapt the existing filters to our data, but not move the parameters so far from the pre-trained parameters.

During fine-tuning we can speed up the process by running the model on the GPU. In PyTorch this can be accomplished by using the `.cuda()` command on the model and loss function. For example:

```python
model = model.cuda()
criterion = criterion.cuda()
```

Finally, after training, we would like to save the model so that we can use it in the future without training again. We can use the `model.save(filename)` function to save the model.

For this project, we do not specify the hyper parameters for you to use. It is up to you to choose a good set of hyper parameters. Examples of hyper parameters that can be changed are learning rate, batch size, and number of epochs.

**Data Augmentation**  To achieve higher performance, we can experiment with data augmentations. Data augmentation is the process of slightly perturbing the input images to generate more samples than were originally available. This data augmentation could be image rotation, scale, gray scale transformation, etc.

PyTorch includes transformations useful for data augmentation in the `torchvision.transforms` module. As part of the project, you need to add some of these data augmentation methods to your preprocessing object to achieve greater test accuracy.

**Additional Resources**
- PyTorch tutorials - [http://pytorch.org/tutorials/](http://pytorch.org/tutorials/)
- PyTorch Github - [https://github.com/pytorch/pytorch](https://github.com/pytorch/pytorch)
- TorchVision Github - [https://github.com/pytorch/vision](https://github.com/pytorch/vision)
- PyTorch Discussion Forum - [https://discuss.pytorch.org/](https://discuss.pytorch.org/)

**Task Assignment**

**Deliverables and Submission Instructions**

Each group is assigned a different dataset and model as shown in Table 3. Submit your project as a zipped folder named `GroupNumber_Project2`, where `GroupNumber` is replaced by your assigned group number. The plots and saved models that you generate for this project should be placed in a folder called `results`. Also include in the submitted folder a Readme file which includes: 1) a description of the obtained results and a comparative analysis of the performance of the finetuned network as compared to the baseline network; and 2) a clear description
Table 3: Group tasks for transfer learning in PyTorch and RNN in Tensorflow

<table>
<thead>
<tr>
<th>Group number</th>
<th>Dataset</th>
<th>Models</th>
<th>Datasets</th>
<th>RNN unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Animals</td>
<td>AlexNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Animals</td>
<td>Inception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Animals</td>
<td>ResNet18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Animals</td>
<td>VGG16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Places</td>
<td>AlexNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Places</td>
<td>Inception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Places</td>
<td>ResNet18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Places</td>
<td>VGG16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Household</td>
<td>AlexNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Household</td>
<td>Inception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Household</td>
<td>ResNet18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Household</td>
<td>VGG16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Caltech101</td>
<td>AlexNet</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Caltech101</td>
<td>Inception</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Caltech101</td>
<td>ResNet18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Caltech101</td>
<td>VGG16</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

of the tasks performed by each group member and how each member contributed to this project. There should be only one submission per group. A list of the required deliverables is shown below.

Deliverable: Project 2

Depending on your group number, you are assigned a different dataset and model to perform transfer learning (Table 3).

- Provide the test accuracy and confusion matrices for the considered pre-trained base network by considering its layers as feature extractors for the dataset; name the confusion matrix plot as `conf_feature.png` and submit this plot in the `results` subfolder.

- Provide the test accuracy and confusion matrices for the considered fine-tuned network for the dataset; name the confusion matrix plot as `conf_finetune.png` and submit this plot in the `results` subfolder.

- Submit code for the feature extraction based method named as `feature.py`.

- Submit the code for the fine-tuning based method named as `finetune.py`.

- Submit the code for the fine-tuning based method with data augmentation named as `fineplus.py`.

- Save and submit the trained models `featrmodel.pt`, `finemodel.pt` and `fineplusmodel.pt` corresponding to the feature extraction, fine-tuning, and fine-tuning with data augmentation methods, respectively. Place the saved models in the `results` subfolder.

- Submit a Readme file including a description and comparison of the results as well as the tasks performed by each group member.

The grading rubric for this project is shown in Table 4.
Table 4: Grading rubric

<table>
<thead>
<tr>
<th>Points</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTE:</td>
<td>DO NOT SUBMIT THE DATASET</td>
</tr>
<tr>
<td>25</td>
<td>Working code for feature extraction based method (feature.py)</td>
</tr>
<tr>
<td>25</td>
<td>Working code for fine-tuning based method (finetune.py)</td>
</tr>
<tr>
<td>10</td>
<td>Working fine-tuning code using data augmentation</td>
</tr>
<tr>
<td>15</td>
<td>Test accuracies and confusion matrices for the assigned dataset and model in a subfolder called results</td>
</tr>
<tr>
<td>10</td>
<td>Submit saved models (three in total)</td>
</tr>
<tr>
<td>15</td>
<td>Readme file with description of results and tasks performed by each group member</td>
</tr>
<tr>
<td>Total 100</td>
<td></td>
</tr>
</tbody>
</table>

References


