Project Part 1 - PyTorch and Transfer Learning

ARIZONA STATE UNIVERSITY
SCHOOL OF ELECTRICAL, COMPUTER,
AND ENERGY ENGINEERING,
EEE598: Deep Learning for Media Processing & Understanding

1 Objectives

- Learn PyTorch Basics
- Extract features from pre-trained networks and train SVM classifiers using the features
- Fine-tune the networks to classify on different datasets

2 PyTorch

PyTorch [1] is an open source machine learning library that is particularly useful for deep learning. PyTorch contains auto-differentation, 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 backwards 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.

2.1 Installation

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

```
pip install http://download.pytorch.org/whl/cpu/torch-0.3.1-cp27-
    cp27mu-linux_x86_64.whl
pip install torchvision
```

The first command installs the basic PyTorch package, and the second command installs `torchvision` which includes functions, classes, and models useful for deep learning for computer vision.

2.2 Tutorial

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

3 CIFAR100 Example in PyTorch

Next as an example, we will re-implement the neural network from Lab 4 using PyTorch instead of the library we built. The code for this example is in the included `cifar.py` file.

PyTorch has a module called `nn` that contains implementations of the most common layers used for neural networks. If you look at the code for some layer (for example `linear.py`), you will see that it is similar to our implementation from lab 3. There is a `forward` method and an `__init__` method. There is no need for a `backward` method because PyTorch uses automatic differentiation.

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. Note that we can reuse the pool layer, since this layer has no learnable parameters. Also instead of a ReLU layer, we can use the `F.relu` function in our forward pass. Similarly, instead of a "flatten" layer, we can just use PyTorch's `view` to reshape the data.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Conv2d args: (input depth, output depth, filter size)
        self.conv1 = nn.Conv2d(3, 16, 3)
        self.conv2 = nn.Conv2d(16, 32, 3)
        # Linear args: (# of input units, # of output units)
        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 means shape is inferred
        x = self.fc1(x)
        return x
```

Fit function Next we will implement a fit function. Here we implement the fit function as a class method, similar to lab 3. It is also common to see the code for training be implemented outside of the model class in a separate function. The fit function is very similar to our own fit function from lab 3. First we set an optimization criterion, and 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 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. The optimization step is similar to the `update_params` methods we used in previous labs.

```
    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

                # wrap them in Variable
                inputs, labels = Variable(inputs), Variable(labels)

                # 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.data[0]

            print('[Epoch: %d] loss: %.3f' %
                  (epoch + 1, running_loss / (i+1)))
            running_loss = 0.0
```

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

        print('Accuracy on test images: %d %%' % (
            100 * correct / total))

        return all_predicted
```

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 missclassifications. The included `util.py` includes the function `plot_confus_matrix` which will plot this matrix. Figure 1 shows the confusion matrix for this CIFAR100 example.

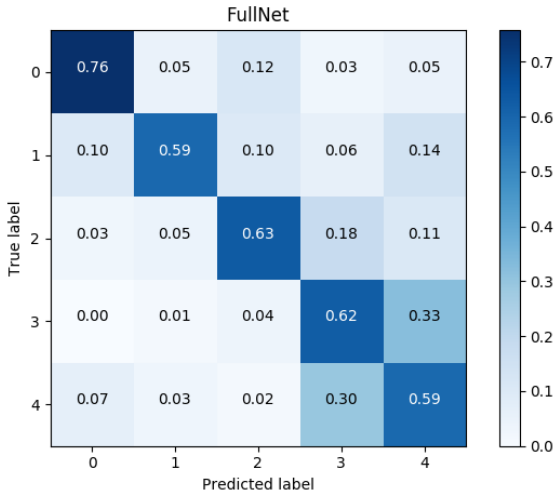


Figure 1: CIFAR100 confusion matrix.

4 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 two types of transfer learning: a feature-extraction based method and a fine-tuning based method. We will be using networks that have been pre-trained on the ImageNet dataset [2], and adapt them for different datasets.

4.1 Datasets

We have collected four datasets for use in this project (Table 1). You will be assigned two of these datasets to work on. A link to the datasets will be available on blackboard. 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 on the full version of 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

Dataset	Description	# Categories	# Train / # Test
Animals	iNat2017 challenge [3]	9	1384 / 466
Faces	Labeled faces in the wild dataset [4]	31	1021 / 439
Places	Places dataset [5]	9	1437 / 367
Household	iMaterialist 2018 challenge [6]	9	1383 / 375

4.2 Base Networks

Table 2 shows the base networks that we will consider in this project. Each group will consider two of the networks (see Section 6 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:

```
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.

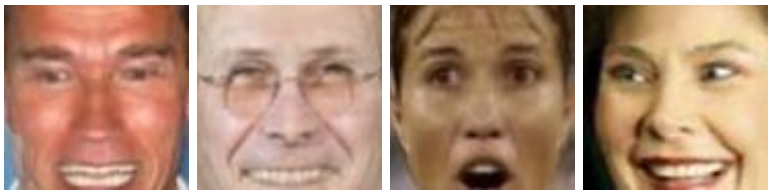
4.3 Pre-processing

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

(a) Household

(b) Animals

(c) Places



(d) Faces

Figure 2: Example images from datasets.

Table 2: Base Networks

Model	Year	Input Size	Last layer input size	PyTorch model
AlexNet [8]	2012	224×224	4096	`torchvision.models.alexnet`
VGG16 [9]	2014	224×224	4096	`torchvision.models.vgg16`
ResNet18 [10]	2016	224×224	512	`torchvision.models.resnet18`
Inception v3 [11]	2015	299×299	2048	`torchvision.models.inception_v3`
DenseNet121 [7]	2017	224×224	1024	`torchvision.models.densenet121`

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 224×224 pixels, but for Inception this is 299×299 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.

4.4 Part 1: Feature Extractor + SVM

First we will use the base network 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 algorithm. In this case we will use an SVM for classification.

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
```

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:

```
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 SVM classification accuracy for the test set. To use the input data in our PyTorch model, we need to wrap it as a PyTorch `Variable`.

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

With the extracted features for each sample, we use Sci-kit learn's SVM model. To review how to use the model, revisit Lab 2. It is up to you to determine the type of SVM and best hyper parameters.

We can assess the accuracy of the SVM 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.

4.5 Part 2: Fine-tuning

The feature extractor + SVM approach already may give decent results for your problem. 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 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.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. 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:

```
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.

5 Additional Resources

- PyTorch Documentation - http://pytorch.org/docs/stable/index.html
- TorchVision Documentation - http://pytorch.org/docs/master/torchvision/index.html
- PyTorch tutorials - http://pytorch.org/tutorials/
- PyTorch Github - https://github.com/pytorch/pytorch

- TorchVision Github - https://github.com/pytorch/vision
- PyTorch Discussion Forum - https://discuss.pytorch.org/

6 Deliverable

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

Table 3: Group tasks

Group number	Dataset	Models
1	Animals	AlexNet, VGG16
2	Animals	AlexNet, Inception
3	Animals	AlexNet, ResNet18
4	Animals	VGG16, Inception
5	Animals	VGG16, ResNet18
6	Animals	Inception, ResNet18
7	Places	AlexNet, VGG16
8	Places	AlexNet, Inception
9	Places	AlexNet, ResNet18
10	Places	VGG16, Inception
11	Places	VGG16, ResNet18
12	Places	Inception, ResNet18
13	Faces	AlexNet, VGG16
14	Faces	AlexNet, Inception
15	Faces	AlexNet, ResNet18
16	Faces	VGG16, Inception
17	Faces	VGG16, ResNet18
18	Faces	Inception, ResNet18
19	Household	AlexNet, VGG16
20	Household	AlexNet, Inception
21	Household	AlexNet, ResNet18
22	Household	VGG16, Inception
23	Household	VGG16, ResNet18
24	Household	Inception, ResNet18

Deliverable: Project Part 1

- Provide the test accuracy and confusion matrices for both considered networks as feature extractors
- Provide the test accuracy and confusion matrices for both considered fine-tuned networks
- Submit the two trained models saved using the `model.save` function
- Submit code for both the feature extraction based method and the fine-tuning based method

Submit your project as a folder named `GROUP_NUMBER_PROJECT_PART1` and zip the folder for submission. The grading rubric for this project is shown in Table 4. There are no unit tests for the

project, but we will grade your code by attempting to run your code. There should be only one submission per group.

Table 4: Grading rubric

Points	Description
40	Working code for feature extraction based method (`feature.py`)
40	Working code for fine-tuning based method (`finetune.py`)
10	Test accuracies and confusion matrices for both models
10	Submit saved models
Total 100	

References

[1] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[3] "iNaturalist challenge at FGVC 2017." https://www.kaggle.com/c/inaturalist-challenge-at-fgvc-2017. Accessed: 2018-04-11.

[4] E. Learned-Miller, G. B. Huang, A. RoyChowdhury, H. Li, and G. Hua, "Labeled faces in the wild: A survey," in *Advances in face detection and facial image analysis*, pp. 189–248, Springer, 2016.

[5] B. Zhou, A. Lapedriza, J. Xiao, A. Torralba, and A. Oliva, "Learning deep features for scene recognition using places database," in *Advances in neural information processing systems*, pp. 487–495, 2014.

[6] "iMaterialist challenge at FGVC 2018." https://www.kaggle.com/c/imaterialist-challenge-furniture-2018. Accessed: 2018-04-11.

[7] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2017.

[8] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[9] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *International Conference on Learning Representations*, 2014.

[10] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *IEEE International Conference on Computer Vision (CVPR)*, 2016.

[11] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," *IEEE International Conference on Computer Vision (CVPR)*, pp. 1–9, 2015.