Hacettepe University

Department of Computer Engineering

BBM418 - Computer Vision Laboratory

Programming Assignment 1

Sevda Sayan
21527305
sevdasayan@cs.hacettepe.edu.tr

May 2019

Contents

1 Introduction

In this assignmnet, we are expected to learn how to do a transfer learning by using CNNs and how to train a network for a specifc problem. For achieving this, pre-trained VGG-16 model and images from a scene dataset are used.

What is VGG-16 model?

VGG16 is a convolutional neural network model. The model achieves 92.7 percent top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. ImageNet is a dataset of over 15 million labeled high-resolution images belonging to roughly 22,000 categories.

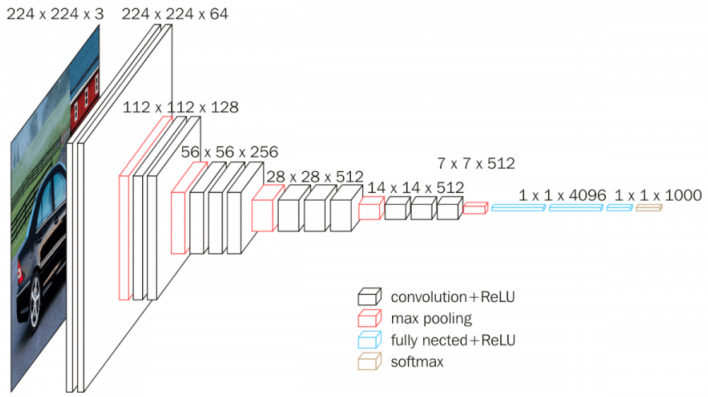

Figure 1: VGG16

Three Fully-Connected (FC) layers follow a stack of convolutional layers (which has a different depth in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

2 Implementation Details

Data augmentation is a good practice for the train set. We simultaneously prepare the images for our network and apply data augmentation to the training set. Each model will have different input requirements, but if we read through what Imagenet requires, we figure out that our images need to be 224x224 and normalized to a range. So train, validation and test data has been randomly cropped to 224x224 and randomly flip it horizontally.

```
"train": transforms.Compose([
    transforms.RandomResizedCrop(224),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
]),
"validation": transforms.Compose([
```

```
dataloaders = {
    x: torch.utils.data.DataLoader(
        image_datasets[x], batch_size=8,
        shuffle=True, num_workers=4
    )
    for x in ["train", "validation", "test"]
}
```

Figure 2: Data Loading

After load data set, we have 4000 images under train, 250 images under validation and 250 images under test as seen below. When we look at classes, there are ten classes under train. The end result of passing through these transforms are tensors that can go into our network.

```
Loaded 4000 images under train
Loaded 250 images under validation
Loaded 250 images under test
Classes:
['airport_inside', 'bar', 'bedroom', 'casino', 'inside_subway', 'kitchen', 'livingroom', 'restaurant', 'subway', 'warehouse']
```

Figure 3: Dataset

2.1 Part 1 - Image Representation

In this part we are expected to extract every image's VGG16 model and get the vectors of FC-7 after ReLU as features for each image in the dataset to obtain the 1x4096 vectors for each image by using a pre-trained network which is trained on ImageNet dataset.

```
model = models.vgg16(pretrained=True)
for param in model.parameters():
    param.requires_grad = False
model.classifier= nn.Sequential(*list(model.classifier.children())[:-2])

use_gpu = torch.cuda.is_available()
if use_gpu:
    model.cuda()

calculate_svm(model)
```

Figure 4: Getting the vectors of FC-7

For getting FC-7 vector, I removed last two layer from my model's classifier. After perform it over an image, I got 1x4096 response for every image and band them together. After obtain pre-trained VGG-16 model from torch library that is trained on ImageNet, this model used for calculating class based/total accuracies by using one-vs-the-rest multi-class linear SVM for classifcation.

```
def calculate_svm(model):
    model.eval()
    .
    .
    .
    print('Accuracy : {:.2f}'.format(100*clf_linear.score(feat_test,feat_classes_test)))
    return (feat_train, feat_classes_train, feat_test, feat_classes_test, clf_linear)


vgg16 = models.vgg16(pretrained=True)
for param in vgg16.features.parameters():
    param.requires_grad = False
vgg16.classifier = nn.Sequential(*list(vgg16.classifier.children())[:-2])

result_arrays = calculate_svm(vgg16)
```

Figure 5: Calculate_SVM Function

Calculate_SVM() function does this job for us. It runs the network in evaluation mode so any weights are not training. It returns four arrays named feat_train, feat_classes_train, feat_test, feat_classes_test and linear svc that fitted with images and labels of them.

Average accuracy is calculated by using build in function named score() from sklearn.svm.LinearSVC library. Eventually, average accuracy has found **72.80 percent**.

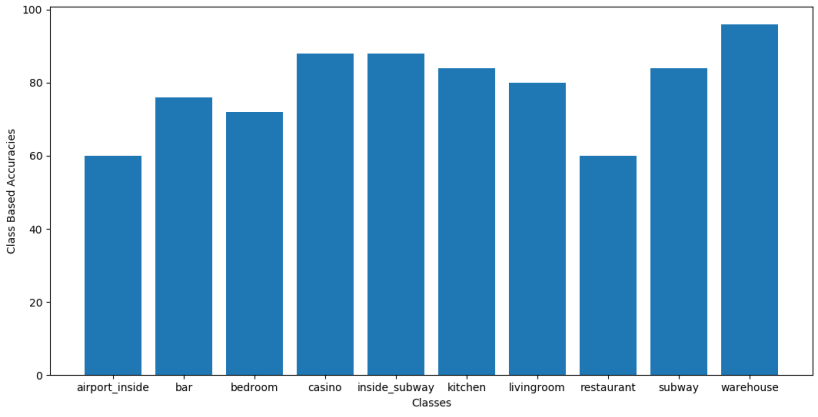

Figure 6: Class Based Accuracies

When we look at the class based accuracies, topmost score is warehouse with 96.0 percent, second high accuracies is casino and inside_subway with 88.0 percent. Based on these results, obtained accuracies are usually can be called high.

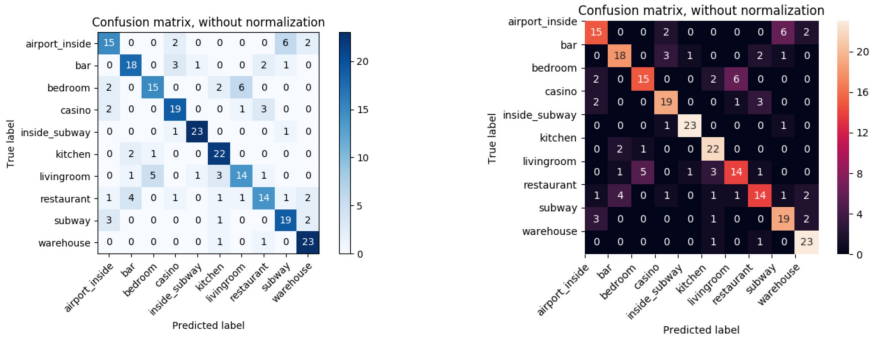

Figure 7: Confusion Matrix

Confusion Matrix is a performance measurement for machine learning classification problem where output can be two or more classes. Well, we have ten classes, and a table with 10 different combinations of predicted and actual values has shown above.

2.2 Part 2 - Fine-tuning

Purpose of this part is observing effects of fine tuning, number of epochs, batch size and learning rate.

What is fine-tuning? Why should we do this?

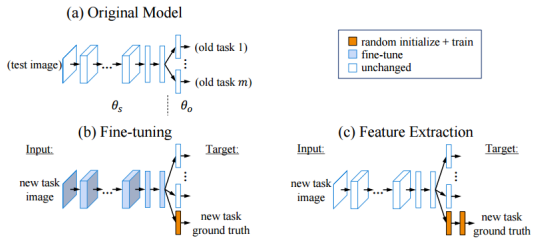

Figure 8: Fine Tuning

Neural networks has a huge number of parameters, often in the range of millions. Training a network on a small dataset (one that is smaller than the number of parameters) greatly affects the network's ability to generalize. Fine tuning seeks to increase the effectiveness or efficiency of a process or function.

Provided that our dataset is not drastically different in context to the original dataset (e.g. ImageNet), the pre-trained model will already have learned features that are relevant to our own classification problem.

Why do we freeze the rest and train only FC layers?

Main purpose underlying it is increase training speed by freezing layers. We'll train only the very last few fully-connected layers. Initially, we freeze all of the model's weights. When the extra layers are added to the model, they are set to trainable by default (require_grad=True). For the VGG-16, we're only changing the very last original fully-connected layer.

Once we have the gradients, we use them to update the parameters with the optimizer. The optimizer is Adam, an efficient variant of gradient descent that generally does not require hand-tuning the learning rate. With the pre-trained model, the custom classifier, the loss, the optimizer, and most importantly, the data, we're ready for training.

The loss and accuracy both for training and validation set

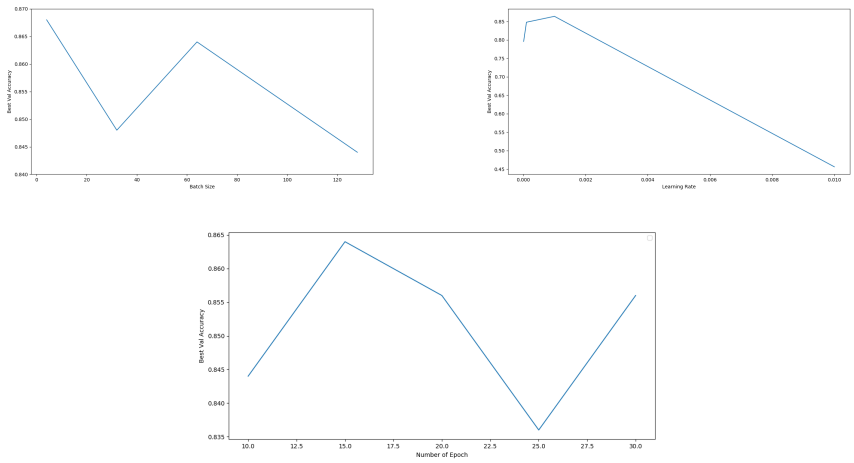

Figure 9: The loss and accuracy against Epoch, Learning Rate and Batch Size

When we look at batch size against best accuracy graph, we can clearly see that it didn't affect our results a considerable extent. When batch size change with values 4, 32, 64 and 128 our best accuracy changed on interval 84 percent and 87 percent.

Otherwise when we look at the number of epoch against best accuracy graph, it again didn't affect our results a considerable extent. When epoch size change with values 10, 15, 20, 25 and 30 our best accuracy changed on interval 83 percent and 86 percent.

Lastly when we look at the learning rate against best accuracy graph, it indicated more change against batch size. When learning rate change with

values 0.01, 0.001, 0.0001 and 0.00001 our best accuracy changed on interval 45 percent and 86 percent.

To see the usefulness of early stopping, we can look at the training curves showing the **training and validation losses and accuracy.**

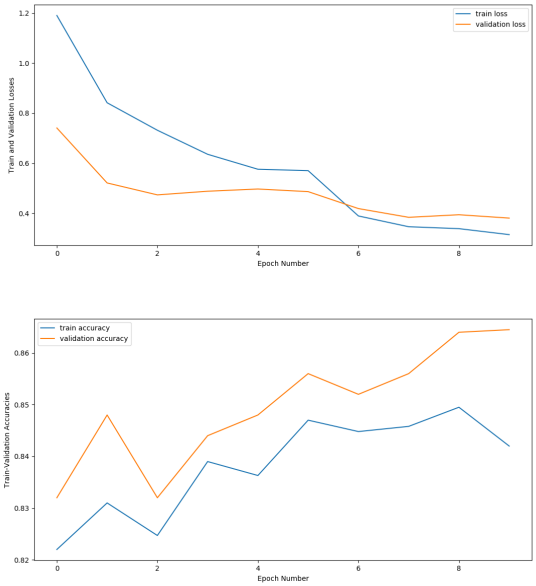

Figure 10: Epoch Number Against Train Loss/Acc and Validation Loss/Acc

As expected, the training loss only continues to decrease with further training. Validation loss, on the other hand, achieves an base furthermore plateaus. At a certain epoch, there is no come back (or indeed going a negative return) on further preparation. This model will best start on remember those preparing information and won't have the capacity will sum up testing information.

Should Accuracy Increase and Train/Validation Loss Decrease After Every Epoch?

It depends on dataset and architecture. But, in a perfect world one would expect the accuracy to increase. If the accuracy starts to decrease, it would be that network is overfitting. We can stop the learning just before network reaches that point which is called early stopping or take other steps to avoid overfitting problem. Overfitting can be identified by looking at validation parameters, like loss or accuracy. These parameters usually stop improving and after begin to decrease. On the other hand, the training parameters improves because the model wants to find the best fit for training data. There are several ways to reduce overfitting. Getting more training data is the best solution. But, due

to limited time we might not have this possibility every time. In addition, we can reduce the overfitting by lowering capacity of the model to memorize the training data. In addition lowering capacity of the model to memorize the training data is another way to decrease possibility of the overfitting problem. By this way, model will focus on the more relevant attributes in data so that generalization on the results will be better.

Top-1 and Top-5 Accuracy of the Test Set

Epoch / Batch	Avg. Loss	Avg. Accuracy
Epoch=10 Batch=4	0.1157	0.8760
Epoch=10 Batch=8	0.0564	0.8540
Epoch=10 Batch=32	0.0140	0.8290
Epoch=15 Batch=4	0.1130	0.7490
Epoch=15 Batch=8	0.0565	0.7160
Epoch=15 Batch=32	0.0147	0.7080
Epoch=20 Batch=4	0.0966	0.9000
Epoch=20 Batch=8	0.0505	0.8920
Epoch=20 Batch=32	0.0121	0.8760

Figure 11: Losses and Accuracies against Epoch and Batch

As a result, our top-1 accuracy is 90 percent and it occurs when epoch is 20 and batch size is 4. Additionally our top-5 accuracies are 90, 89.2, 87.6, 87.6 and 85.4 percent and they occurs when epoch is 20 and batch size is 4, when epoch is 20 and batch size is 8, when epoch is 10 and batch size is 4, when epoch is 20 and batch size is 32 and when epoch is 10 and batch size is 8.

2.3 Part 3

Transfer learning was a process where you took the first n layers from a pre-trained model, added on your own final layers for your task, and then fine tuning was where you did NOT freeze the weights from layers you transferred from the pre-trained model, but instead allowed them to update with a very low learning rate. Also this method resulted in better generalisation, and results in general, than freezing the weights from the transferred layers.

In this part, we will compare our results for both fine-tuned and pretrained image weights. My obtained graphs results are indicated below.

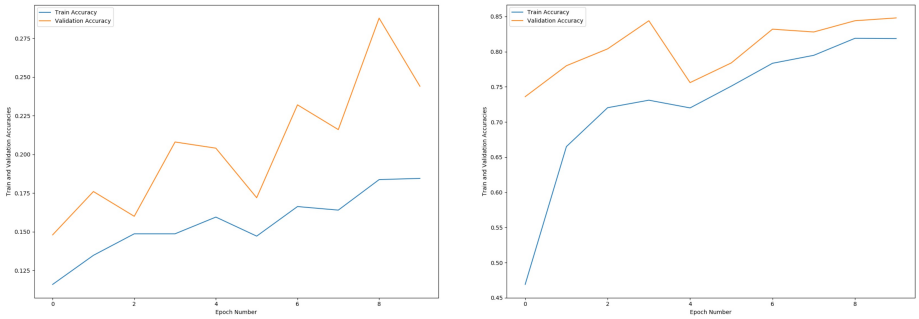

Figure 12: Epoch Number Against Train and Validation Accuracies when fine-tuning and pretrained=true

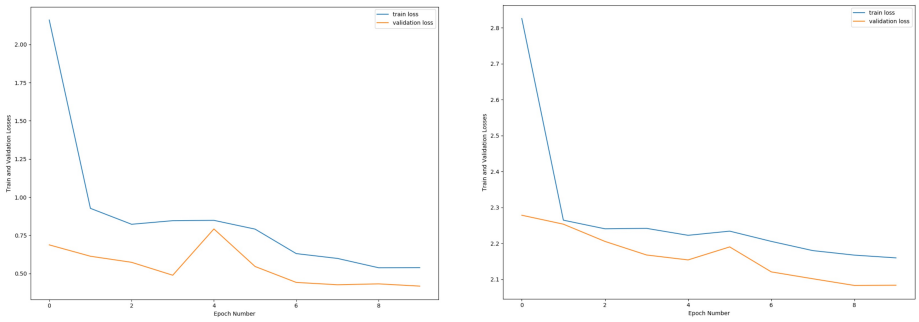

Figure 13: Epoch Number Against Train and Validation Losses when fine-tuning and pretrained=true

As we expected, whether or not model is pretrained affects our train/validation losses and accuracies significiantly. When we use pretrained Imagenet weights, train losses changes on inverval 0.53 and 0.95 accuracies changes on interval 0.4 and 0.82. Validation losses changes on inverval 0.7 and 0.4, accuracies changes on interval 0.72 and 0.85.

Otherwise when we use not-pretrained Image weights train losses changes on inverval 2.85 and 2.15 accuracies changes on interval 0.1 and 0.18. Validation losses changes on inverval 2.3 and 2.0, accuracies changes on interval 0.14 and 0.25.

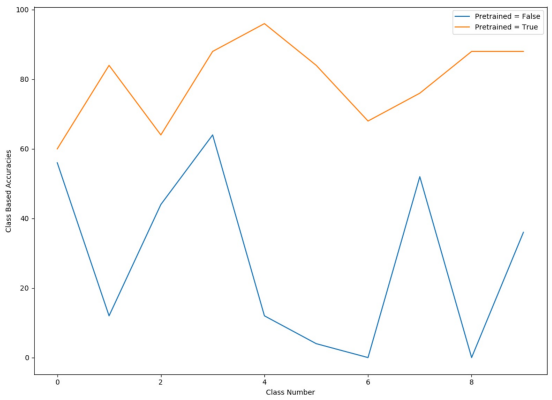

Figure 14: Class based accuracies when fine-tuning and pretrained=true

Lastly when we compare class based accuracies on both cases, we can clearly observe that fine-tuned network accuracies far more than the other.