520.638 Deep Learning
Homework 3 - Network Fine-Tuning

Nicholas Hinke

April 12, 2022

1 Part 1: Written Questions

1. *Briefly describe the idea behind a DenseNet? Discuss its advantages and disadvantages.*

   Partially inspired by the motivation behind residual blocks within ResNet, DenseNet utilizes what are known as “dense blocks” in order to achieve a very interesting flow of information through the network. In essence, each layer within a dense block takes in the outputs of all the preceding layers as an input, a phenomenon which encourages feature reuse. Due to the increased number of connections, DenseNets come with the obvious cost of increased computational complexity when compared to a standard feed-forward network of the same size. For the same reason, DenseNets can also be more prone to overfitting to training data without adequate regularization. Despite these disadvantages, however, DenseNets are very useful as they are typically able to substantially outperform other networks on small datasets due to the greatly increased number of connections.

2. *Briefly describe the idea behind an inception module? Discuss its advantages and disadvantages.*

   When designing convolutional layers within a CNN, it can take a very substantial amount of time to find the optimal filter/kernel sizes (*i.e.* you may have to perform cross-validation using many different combinations of filter sizes in each layer). The idea behind inception modules is to apply several different convolution kernels of varying size, and to let the network decide which ones are optimal through the parameters learned from backpropagation. The problem with this approach is its extreme computational cost, as it is much more expensive to apply many (*e.g.* four) different filters at a given layer than only one. To combat this issue, the creators of GoogleNet popularized the idea of using bottleneck layers (1x1 convolutions to reduce dimensionality) with inception modules to improve the computational efficiency of the combined inception block.

3. *Describe at least three loss functions that can be used to train an autoencoder network.*

   Since autoencoders are essentially a method of unsupervised reconstruction for representation learning, a type of “reconstruction loss” should be used as the loss function when training an autoencoder. As such, any measure of dissimilarity between representations can be incorporated into a loss function for reconstruction loss. Three such examples are Mean-Squared-Error (MSE) loss (based on the l_2-norm), l_1-norm loss, and cosine angle similarity loss.

4. *Briefly describe a sparse autoencoder.*

   Sparse autoencoders provide an example of how a regularization technique can be applied to autoencoders. Sparse autoencoders utilize more hidden units than input units, but drop many of the hidden units during training so that only a small number of them may be active at the same time (similar to the technique of dropout used for regularization). Moreover, sparsity within the autoencoder can be achieved by adding an additional regularization term (*e.g.* l_1-norm of parameters) to the reconstruction loss.

5. *Briefly describe a de-noising autoencoder.*

   The idea behind a de-noising autoencoder is to make the autoencoder network work harder (and therefore learn more robust representations) by first adding noise (*i.e.* small perturbations) to the inputs that are fed as inputs to it. Thus, the network is forced to learn how to remove the perturbations in the input (*i.e.* de-noise it) in order to find good, robust representations. As a result, de-noising autoencoders can be very useful for recovering the corresponding "clean" inputs.

6. *What are the differences between segmentation, semantic segmentation and instance segmentation?*

   Given an image as the input, the idea behind "segmentation" is to delineate which pixels in the image correspond to "coherent parts" *without* attempting to understand what is represented within each part (*i.e.* not performing classification on them). "Semantic segmentation" takes this idea a step further by delineating which pixels in the image correspond to semantically meaningful parts (*e.g.* different classes); in other words, the essence of semantic segmentation can be thought of as combining segmentation with classification. Finally, "instance segmentation" combines the idea of semantic segmentation with that of object detection, by identifying every unique instance of an object corresponding to each class within the segmented image.

7. *Describe at least three approaches that we discussed in class for object detection.*

   First, the classical approach for object detection involves a sliding window detector, where small cropped portions of the image are first found by "sliding" a window of variable size across the entire image, and are then passed into a CNN to determine whether or not the object is present in that window. Second, the approach popularized by the RCNN series of networks involves utilizing some method of region proposal (*e.g.* selective search algorithm or a separate region proposal network) to find many regions of interest (RoI) within an image, and then utilizing a classifier (*e.g.* CNN, SVM) to determine whether or not the object is present within a given region. Third, the approach designed by the creators of the YOLO framework utilizes predetermined "anchor boxes" to perform object detection. Within this framework, an entire image is taken as input and first split into a grid where each grid cell contains several anchor boxes (each essentially acting as a candidate bounding box). Then, a single CNN can be used to determine whether or not an object is present within each anchor box.

8. *Briefly describe non-max suppression.*

   When performing object detection using any of the approaches described above, there will often be many object detections (*i.e.* overlapping bounding boxes containing the object) that correspond to the same object. The idea behind non-max suppression is to determine a method to pick the detection window that best represents the object, and to suppress the rest. Practically, this is done by selecting the detection window with the highest intersection over union (IOU) and suppressing the others; consequently, the detection window that draws the tightest bounding box around the detected object will be chosen.

9. *Describe the differences among RCNN, Fast-RCNN and Faster-RCNN.*

   RCNN, Fast-RCNN, and Faster-RCNN all rely on the same motivating principal of finding regions of interest (which are found through some method of region proposal) and extracting useful salient features through the use of a CNN, and then using some classification method to determine whether or not an object is present. RCNN was the first of the three networks, and uses a selective search algorithm based on segmentation to determine the regions of interest within the image *before* passing each region through a CNN. It then uses a support vector machine (SVM) to perform classification on the extracted features from the CNN. This network worked well but was extremely slow, thus providing the inspiration for Fast-RCNN and Faster-RCNN. Fast-RCNN came next, and first passes the images through a CNN to extract features before performing region proposal via the selective search algorithm in feature space (using RoI pooling). It then does classification on the RoI feature vector using a CNN. Fast-RCNN was substantially faster than the original RCNN (inference times of less than 2.5 seconds compared to around 45 seconds), but still was not fast enough to be used in most real-time applications. Finally, Faster-RCNN was constructed, and utilizes a completely separate region proposal CNN rather in order to eliminate the use of the time-expensive selective search algorithm. After identifying regions using the region proposal network, classification is then done using a CNN to determine whether or not an object is present. Faster-RCNN was able to reduce inference time to less than 0.25 seconds, and consequently can be used for some real-time applications.

10. *Briefly describe the YOLO framework for object detection.*

    In contrast to other object detection methods, the YOLO framework ("You Only Look Once") takes the entire image as an input rather than selected regions. By first splitting the entire image into a grid and utilizing predetermined "anchor boxes" (pre-selected candidate bounding boxes within each grid cell), the YOLO framework can use a single convolutional network to classify whether or not an object is present in each anchor box. Finally, non-max suppression is performed in order to obtain the final bounding boxes around each object within the image. It is also worth noting that the anchor box method is by far the most popular method for object detection in research today, as it allows for much faster image processing (*e.g.* YOLO can run at up to 45 frames per second). Additionally, despite all of YOLO's successes, it sometimes struggles with small objects within images.

11. *How can we measure the performance of an object detector?*

    As briefly described above within the context of non-max suppression, the intersection over union metric (IOU) is an easy way to compare how much overlap there is between the ground truth object bounding box and the predicted bounding box from the object detection model (*i.e.* how accurate is the prediction). This IOU metric can be computed simply by dividing the area where the two bounding boxes intersect (*i.e.* the intersection) by the total area covered by either bounding box (*i.e.* the union). Moreover, there are many other metrics that are useful in comparing the performance of object detectors, such as the mean average precision (mAP) which utilizes the precision and recall for each detector over a range of confidence thresholds.

2 Part 2: Verification on LFW Dataset

Within this section, pre-trained AlexNet and VGG-16 models were re-purposed for the task of verification on the Labeled Faces in the Wild (LFW) dataset. Note that the two models and dataset are publicly available at:

- `https://github.com/BVLC/caffe/tree/master/models/bvlc_alexnet`
- `http://www.robots.ox.ac.uk/~vgg/research/very_deep/`
- `http://vis-www.cs.umass.edu/lfw/`

When performing this task, it was desired to observe the potential performance improvements that could be realized after fine-tuning the pre-trained networks. As such, the performance of the pre-trained models was first evaluated on the LFW test dataset before any fine-tuning was attempted. Since both pre-trained models were trained for 1000-class classification on the ImageNet dataset, both models were already quite good at extracting useful features from their convolutional layers. Fortunately, we can reuse a lot of the information from these convolutional layers (as feature extractors) in our fine-tuned models, but it is highly likely that the model performances could be improved significantly by retraining the fully-connected classification layers (since most of the 1000 ImageNet classes are rather unrelated to human faces). Thus, one approach to fine-tuning the pre-trained models would be to freeze all of the parameters in the convolutional layers and only update the parameters in the fully-connected layers. While this will certainly improve the resulting model performances, this approach ignores the fact that while the convolutional layers have been trained to extract features which are applicable to a very broad range of object classes, they may or may not always be particularly useful (or optimal) for distinguishing (or verifying) human faces. Additionally, since the models were trained for classification and we are interested in the task of verification, it is highly likely that there is potential to improve performance by updating the parameters associated with feature extraction. As such, our approach to fine-tuning will involve two parts in series, each for a specified number of epochs during training:

1. Freeze all of the parameters in the convolutional layers since they are used for feature extraction, and only update the parameters in the fully-connected layers (using backpropagation).

2. Unfreeze the parameters in the convolutional layers and update all of the parameters in the network (using backpropagation).

Due to our utilization of the pre-trained models (which are already quite skilled at feature extraction), we are able to get away with training these large networks (on a comparably small dataset) for a relatively small number of epochs. However, since we are performing the task of verification rather than classification, we must be deliberate in the way we go about training the models and design our loss function accordingly. In order to fine-tune one of the pre-trained networks, we will first pass two images as two separate inputs through it. Then, after obtaining the two 1000×1 output vectors, we will compute the Euclidean distance between the two outputs (*i.e.* the 2-norm of the difference of the output vectors) as a measure of the "dissimilarity" between the two images. We will then normalize that dissimilarity measure such that two identical images will have a dissimilarity of 0, and the two "most dissimilar" images in the dataset will have a dissimilarity of 1. Practically speaking, this can be done by dividing every dissimilarity measure by the maximum dissimilarity measure. Since a dissimilarity measure of 0 corresponds to identical images (and consequently, two faces belonging to the same person), the corresponding label will be 0 if the two images correspond to the same person and 1 if the two images are of different people.

At this stage, we have taken two input images and computed their dissimilarity using their 1000×1 model output vectors, as well as generated a label of whether or not the images correspond to the same person. As such, it is conceivable that we could train the network to produce 1000×1 outputs that will have very low dissimilarity measures (close to 0) if two images are of the same person, and very high dissimilarity measures (close to 1) if two images are of different people. With that goal in mind, we will use binary cross entropy as our loss function in order to encourage this desired behavior. Combining this approach with our aforementioned fine-tuning strategy should yield relatively substantial performance improvements after taking advantage of the pre-trained model parameters.

When evaluating the models (either the pre-trained or fine-tuned versions), we can employ the same strategy as we did during training to get dissimilarity measures between a pair of test images. We can then set a threshold (*e.g.* dissimilarity $< 0.3 \implies$ same person) to make predictions of whether or not the two images correspond to the same person. Following this procedure, we can compute the true and false positive rates (TPR and FPR) using a range of thresholds in order to produce a receiver operating curve (ROC). Once we have obtained the ROC, we can then find the area under the ROC curve (AUC) which can be used as a scale and threshold-invariant metric for comparing model performance (where AUC=0.0 if model is wrong 100% of the time, and AUC=1.0 if model is correct 100% of the time).

After evaluating the various models, each model's performance was summarized by plotting an ROC curve with the resulting AUC listed on it. The results will be summarized in the following subsections. It should also be noted that throughout this section, all code used to complete these problems was written in Python utilizing the PyTorch framework (and is publicly available at `https://github.com/nhinke/deeplearning-repo/tree/master/Homework/hw3`) with the use of the following libraries:

- **os**
- **math**
- **numpy**
- **torch**
- **torchvision**
- **sklearn.metrics**
- **matplotlib.pyplot**

2.1 AlexNet Verification Performance

As described in detail in the previous section, each model's performance was evaluated by first making predictions on pairs of images in the LFW test dataset, and then plotting ROC curves and computing the corresponding AUC. This performance was then repeated for each of the four models (pre-trained AlexNet, fine-tuned AlexNet, pre-trained VGG-16, and fine-tuned VGG-16) in order to observe the performance improvements realized after fine-tuning the networks.

For each model, this performance evaluation was completed using two different versions of the LFW test dataset–consequently, there will be two provided output plots for each model. During the second and more comprehensive evaluation, the model was tested using every possible pair of the 3708 images (6872778 pairs) in the original (not funneled) LFW test dataset (known as the "LFWPeople" dataset). During the first evaluation, however, the model was tested against the smaller "LFWPairs" dataset (also original, not funneled) where each of the 1000 samples contains a pair of images and corresponding verification label.

2.1.1 Pre-trained AlexNet Results

The resulting performance plots for the pre-trained AlexNet model on both versions of the LFW test dataset are provided below:

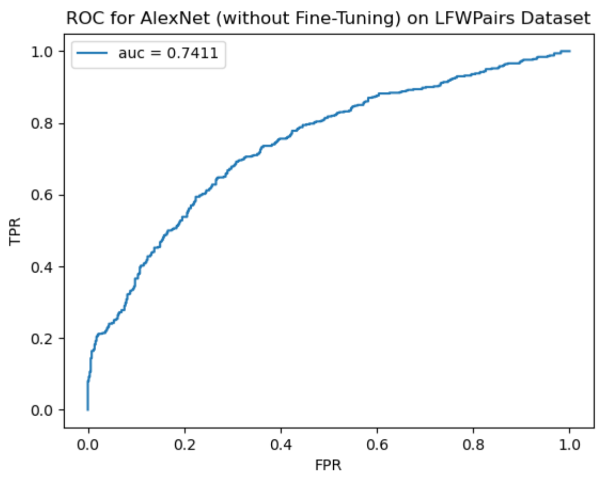


Figure 1: Pre-trained AlexNet Verification Performance on LFWPairs

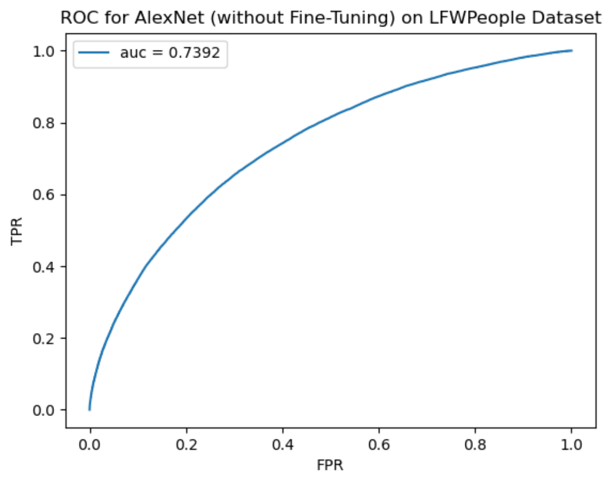


Figure 2: Pre-trained AlexNet Verification Performance on LFWPeople

2.1.2 Fine-tuned AlexNet Results

The resulting performance plots for the fine-tuned AlexNet model on both versions of the LFW test dataset are provided below:

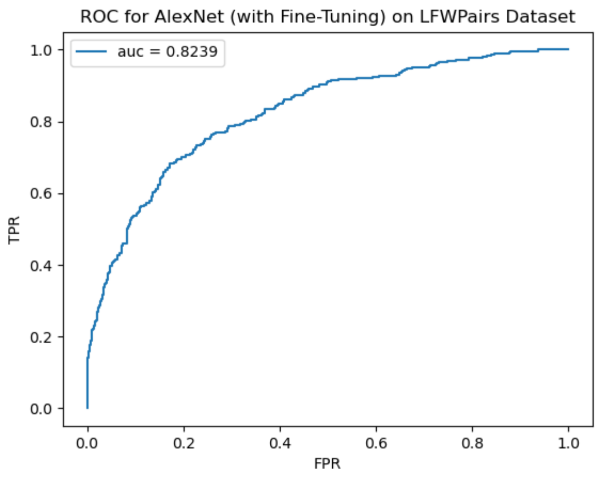


Figure 3: Fine-tuned AlexNet Verification Performance on LFWPairs

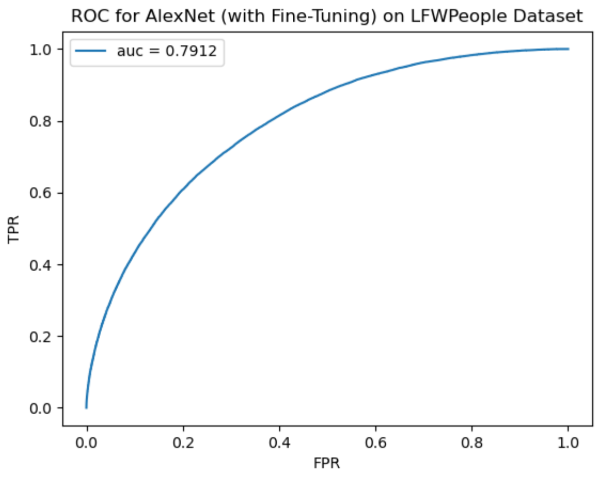
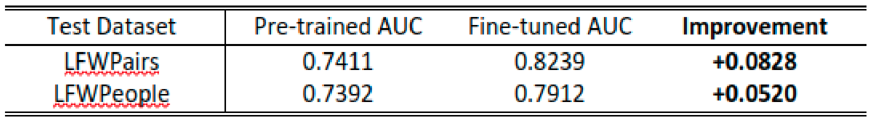


Figure 4: Fine-tuned AlexNet Verification Performance on LFWPeople

2.1.3 AlexNet Model Analysis

The performance increases that were realized as a result of fine-tuning the pre-trained AlexNet model are provided in the table below:



Test Dataset	Pre-trained AUC	Fine-tuned AUC	**Improvement**
LFWPairs	0.7411	0.8239	**+0.0828**
LFWPeople	0.7392	0.7912	**+0.0520**

Table 1: Summary of AlexNet Verification Performances

As can clearly be observed from the results in the table above, fine-tuning the pre-trained AlexNet model yielded a rather significant performance increase, as expected (see previous discussion on fine-tuning strategy and loss function design for more details). It is also worth noting that even the raw pre-trained model (without any fine-tuning) was still quite successful when applied to this verification task, thus demonstrating the true power and generality of the extracted features. Finally, for completeness, the following hyperparameters were used when fine-tuning the pre-trained model using stochastic gradient descent (SGD) on a NVIDIA GeForce RTX™ 3060:

- batch size = 22
- momentum = 0.9
- learning rate = 0.0001
- epochs training classifier (pt.1) = 5
- epochs training whole network (pt.2) = 15

2.2 VGG-16 Verification Performance

As described in detail in the previous sections, each model's performance was evaluated by first making predictions on pairs of images in the LFW test dataset, and then plotting ROC curves and computing the corresponding AUC. This performance was then repeated for each of the four models (pre-trained AlexNet, fine-tuned AlexNet, pre-trained VGG-16, and fine-tuned VGG-16) in order to observe the performance improvements realized after fine-tuning the networks. Since the identical procedure was followed as previously described for the AlexNet models, see above for more details.

2.2.1 Pre-trained VGG-16 Results

The resulting performance plots for the pre-trained VGG-16 model on both versions of the LFW test dataset are provided below:

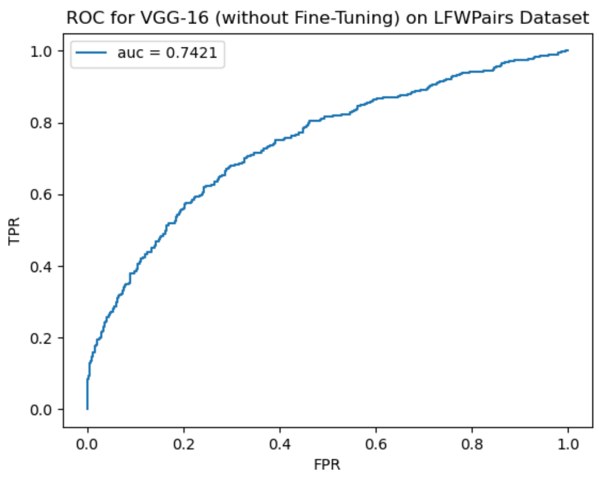


Figure 5: Pre-trained VGG-16 Verification Performance on LFWPairs

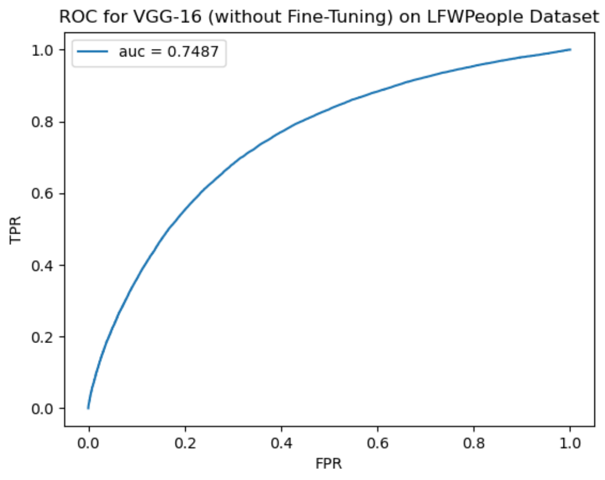


Figure 6: Pre-trained VGG-16 Verification Performance on LFWPeople

2.2.2 Fine-tuned VGG-16 Results

The resulting performance plots for the fine-tuned VGG-16 model on both versions of the LFW test dataset are provided below:

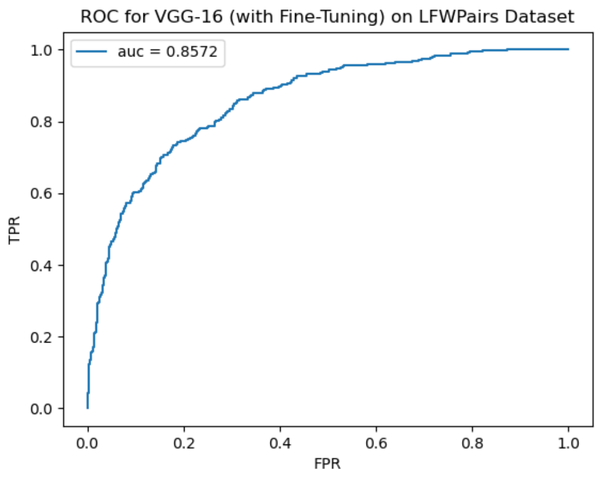


Figure 7: Fine-tuned VGG-16 Verification Performance on LFWPairs

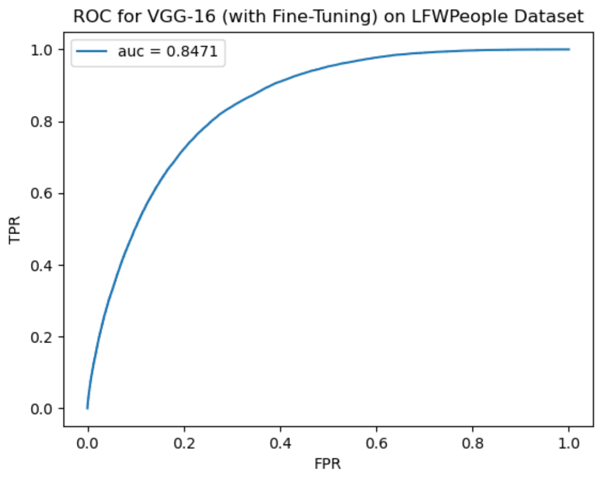
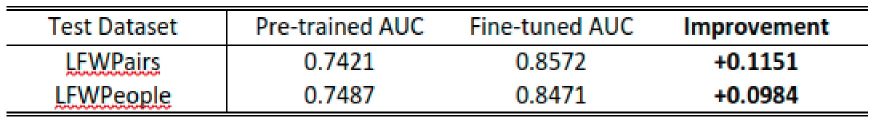


Figure 8: Fine-tuned VGG-16 Verification Performance on LFWPeople

2.2.3 VGG-16 Model Analysis

The performance increases that were realized as a result of fine-tuning the pre-trained VGG-16 model are provided in the table below:



Test Dataset	Pre-trained AUC	Fine-tuned AUC	**Improvement**
LFWPairs	0.7421	0.8572	**+0.1151**
LFWPeople	0.7487	0.8471	**+0.0984**

Table 2: Summary of VGG-16 Verification Performances

As can clearly be observed from the results in the table above, fine-tuning the pre-trained VGG-16 model yielded a rather significant performance increase, as expected (see previous discussion on fine-tuning strategy and loss function design for more details). It is also worth noting that even the raw pre-trained model (without any fine-tuning) was still quite successful when applied to this verification task, thus demonstrating the true power and generality of the extracted features. Finally, for completeness, the following hyperparameters were used when fine-tuning the pre-trained model using stochastic gradient descent (SGD) on a NVIDIA GeForce RTX™ 3060:

- batch size = 4
- momentum = 0.9
- learning rate = 0.0001
- epochs training classifier (pt.1) = 3
- epochs training whole network (pt.2) = 2

Regarding the hyperparameters listed above, it is worth noting that the small batch size was chosen due to memory limits on my GPU. It is also worth mentioning that higher numbers of epochs were also attempted, but yielded inferior performance (unlike that of AlexNet) possibly due to overfitting given the higher complexity of the VGG-16 architecture.