PRACTICAL 2

Due on April 29, 2020 (23:59:59)

Instructions. In Practical 2, you will learn all about the convolutional neural networks. In particular, you will gain a first-hand experience of the training process, understand the architectural details, and familiarize with transfer learning with deep networks. **You are allowed to do this assignment in pairs.** If you wish, you may also do it individually.

Getting set up

In this assignment, you will work on convolutional neural network on artist classification. The starter code and the dataset for Part 1 can be found at the course website. The details about obtaining the data and the pretrained models for Part 2 are explained therein in detail.

Part 1 Convolutional Neural Networks[1]

In this part, you will experiment with a convolutional neural network implementation to perform image classification. The dataset we will use for this assignment was created by Zoya Bylinskii, and contains 451 works of art from 11 different artists all downsampled and padded to the same size. The task is to identify which artist produced each image. The original images can be found in the `art_data/artists` directory included with the practical. The composition of the dataset and a sample painting from each artist are shown in Table 1.

Table 1: Artists and number of paintings in the dataset.

Artist	#	Example	Artist	#	Example
Canaletto	20		Paul Gauguin	37	
Claude Monet	54		Paul Sandby	36	
George Romney	63		Peter Paul Rubens	28	
J. M. W. Turner	70		Rembrandt	40	
John Robert Cozens	40		Richard Wilson	23	
Paul Cezanne	40				

[1]Adapted from a homework developed for MIT 6.867 Machine Learning course, offered by Leslie P Kaelbling, Tomas Lozano-Perez, and Suvrit Sra. Adapted to PyTorch by Sercan Amac.

Figure 1 shows an example of the type of convolutional architecture typically employed for similar image recognition problems. Convolutional layers apply filters to the image, and produce layers of feature maps. Often, the convolutional layers are interspersed with pooling layers. The final layers of the network are fully connected, and lead to an output layer with one node for each of the K classes the network is trying to detect. We will use a similar architecture for our network.

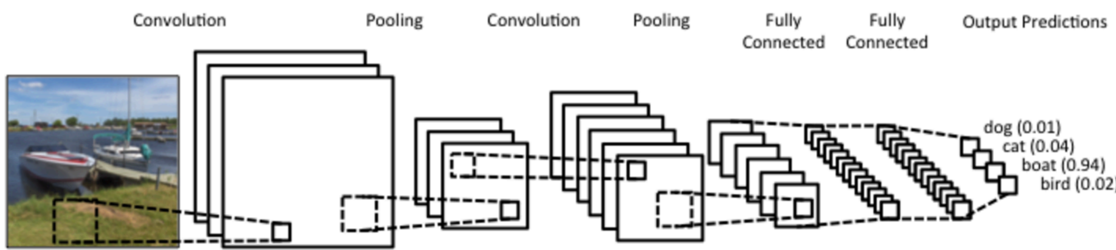

Figure 1: A typical convolutional architecture. Image taken from
http://www.wildml.com/2015/11/understanding-convolutional-neural-networks-for-nlp/

The code for performing the data processing and training the network is provided in the starter pack. You will use PyTorch to implement convolutional neural networks.

In the code, the file `DataLoader.py` creates a dataset from the artists' images by downsampling them to 50x50 pixels, and transforming the RGB values to lie within the range $[-0.5, 0.5]$. The code to train the network can be found in `train.py`[2]. You do not need to understand all of this code to complete the assignment, but you will need to make changes to the hyperparameters and network structure found in the `CnnModel.py` and `run.py`.

1.1 Convolutional filter receptive field

First, it is important to develop an intuition for how a convolutional layer affects the feature representations that the network learns. Assume that you have a network in which the first convolutional layer applies a 5×5 patch to the image, producing a feature map Z_1. The next layer of the network is also convolutional; in this case, a 3×3 patch is applied to the feature map Z_1 to produce a new feature map, Z_2. Assume the stride used in both cases is 1. Let the *receptive field* of a node in this network be the portion of the original image that contributes information to the node (that it can, through the filters of the network, "see"). What are the dimensions of the receptive field for a node in Z_2? Note that you can ignore padding, and just consider patches in the middle of the image and Z_1. Thinking about your answer, why is it effective to build convolutional networks deeper (i.e. with more layers)?

1.2 Run the Pytorch ConvNet

Study the provided code in the SimpleCNN class in `CnnModel.py` and the parameters in `run.py`. Answer the following questions about the initial implementation:

- How many layers are there? Are they all convolutional? If not, what structure do they have?
- Which activation function is used on the hidden nodes?
- What loss function is being used to train the network?
- How is the loss being minimized?

[2]Note that this code was optimized for simplicity, not best programming practices.

Now that you are familiar with the code, try training your network! This can be accomplished by simply running the command `python run.py`. It should take between 60-120 seconds to train your network for 50 epochs. What is the training accuracy for your network after training? What is the validation accuracy? What do these two numbers tell you about what your network is doing?

1.3 Add pooling layers

We will now add max pooling layers after each of our convolutional layers. This code has already been provided for you; all you need to do is switch the `pooling` flag in the hyperparameters to **True**, and choose different values for the pooling filter size and stride. After you applied max pooling, what happened to your results? How did the training accuracy vs. validation accuracy change? What does that tell you about the effect of max pooling on your network?

1.4 Regularize your network!

Because this is such a small dataset, your network is likely to overfit the data. Implement the following ways of regularizing your network. Test each one individually, and discuss how it affects your results.

- **Dropout.** In PyTorch, this is implemented using the torch.nn.dropout class, which takes a value called the `keep_prob`, representing the probability that an activation will be dropped out. This value should be between 0.1 and 0.5 during training, and 0 for evaluation and testing. An example of how this works is available here. You should add this to your network and try different values to find one that works well.
- **Weight regularization.** You should try different optimizers in here. And you should try different weight decay values for optimizers if it is available.
- **Early stopping.** Stop training your model after your validation accuracy starts to plateau or decrease (so that you do not overtrain your model). The number of steps can be controlled through the `patience` hyperparameter in `run.py`.
- **Learning Rate Scheduling** Learning rate scheduling is an important part of training neural networks. There are a lot of techniques for learning rate scheduling. You should try different schedulers such as `StepLR` , `CosineAnnealing` etc.. You may find this link helpful here.

Discuss your results. Which regularization technique was most effective?

1.5 Experiment with your architecture

All those parameters at the top of `SimpleCNN` still need to be set. You cannot possibly explore all combinations; so try to change some of them individually to get some feeling for their effect (if any). Optionally, you can explore adding more layers. Report which changes led to the biggest increases and decreases in performance. In particular, what is the effect of making the convolutional layers have (a) a larger filter size, (b) a larger stride and (c) greater depth? How does a pyramidal-shaped network in which the feature maps gradually decrease in height and width but increase in depth (as in Figure 2) compare to a flat architecture, or one with the opposite shape?

1.6 Optimize your architecture

Based on your experience with these tests, try to achieve the best performance that you can on the validation set by varying the hyperparameters, architecture, and regularization methods. You can even (optionally)

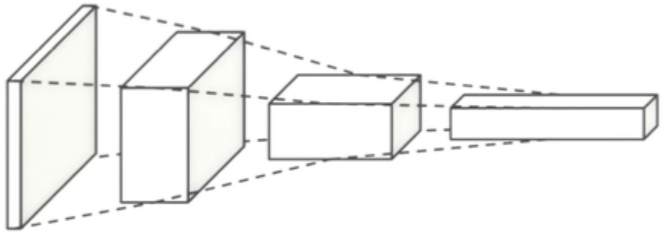

Figure 2: Pyramidal architecture with 3 convolutional layers. Figure was adapted from `https://www.udacity.com/course/deep-learning--ud730`

try to think of additional ways to augment the data, or experiment with techniques like local response normalization layers using torch.nn.LocalResponseNorm or the new weight normalization using the implementation here. Report the best performance you are able to achieve, and the settings you used to obtain it.

1.7 Test your final architecture on variations of the data

Now that you have optimized your architecture, you are ready to test it on augmented data!

In PyTorch data augmentation can be done dynamically while loading the data using what they call transforms. Note that some of the transforms are already implemented in the file `run.py`. You can try other transformations, such as the ones shown in Figure 3 and also try different probabilities for these transformations. You may find this link helpful. Note that the PyTorch data loader refreshes the data in each epoch and apply different transformations to the different instances.

Report your performance on each of the transformed datasets. Are you surprised by any of the results? Which transformations is your network most invariant to, and which lead it to be unable to recognize the images? What does that tell you about what features your network has learned to use to recognize artists' images?

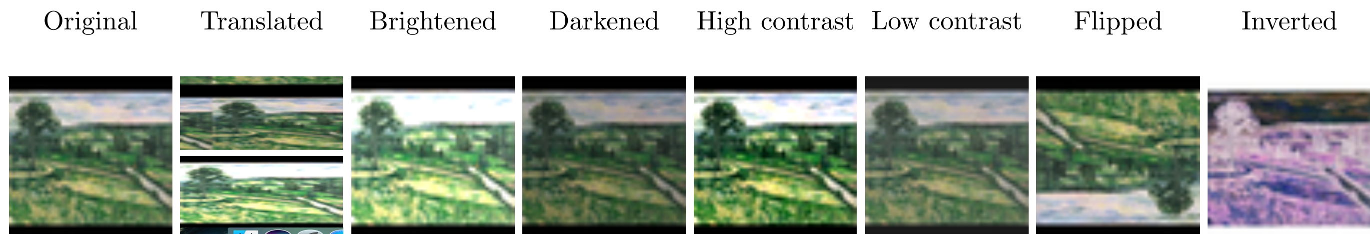

Figure 3: Image transformations used in data augmention.

Part 2 Transfer Learning with Deep Networks[3]

In this part, you will fine-tune AlexNet model pretrained on ImageNet to recognize faces. For the sake of simplicity you may use the pretrained AlexNet model provided in PyTorch Hub. You will work with a subset of the FaceScrub dataset. The subset of male actors is here and the subset of female actors is here. The dataset consists of URLs of images with faces, as well as the bounding boxes of the faces. The format of the bounding box is as follows (from the FaceScrub `readme.txt` file):

[3] Adapted from a homework developed for Toronto CSC321 Introduction to Machine Learning and Neural Networks course, offered by Michael Guerzhoy.

> `The format is x1,y1,x2,y2, where (x1,y1) is the coordinate of the top-left corner of the bounding box and (x2,y2) is that of the bottom-right corner, with (0,0) as the top-left corner of the image. Assuming the image is represented as a Python NumPy array I, a face in I can be obtained as I[y1:y2, x1:x2].`

You may find it helpful to use and/or modify this script for downloading the image data. Note that you should crop out the images of the faces and resize them to appropriate size before proceeding further. Make sure to check the SHA-256 hashes, and make sure to only keep faces for which the hashes match. You should set aside 70 images per faces for the training set, and use the rest for the test and validation set.

2.1 Train a multilayer perceptron

First resize the images to 28×28 pixels. Use a fully-connected neural network with a single hidden layer of size 300 units.

In your report, include the learning curve for the test, training, and validation sets, and the final performance classification on the test set. Include a text description of your system. In particular, describe how you preprocessed the input and initialized the weights, what activation function you used, and what the exact architecture of the network that you selected was. You might get performances close to 80-85% accuracy rate.

2.2 AlexNet as a fixed feature extractor

Extract the values of the activations of AlexNet on the face images. Use those as features in order to perform face classification: learn a fully-connected neural network that takes in the activations of the units in the AlexNet layer as inputs, and outputs the name of the person. In your report, include a description of the system you built and its performance. It is recommended to start out with only using the `conv4` activations. Using `conv4` is sufficient here.

2.3 Visualize weights

Train two networks the way you did in Part 2.1. Use 300 and 800 hidden units in the hidden layer. Visualize 2 different hidden features (neurons) for each of the two settings, and briefly explain why they are interesting. A sample visualization of a hidden feature is shown in Figure 4.

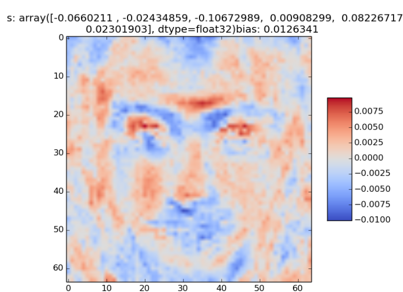

Figure 4: A sample visualization of a hidden feature.

Note that you probably need to use L2 regularization while training to obtain nice weight visualizations.

2.4 Finetuning AlexNet

In Part 2.2, you used the (intermediate) outputs of AlexNet as inputs to a different network. You could view this as adding more layers to AlexNet (starting from e.g. conv4). Modify the classification layer of the AlexNet model so that you have the same output dimensions as in Part 2.2. In your report, include an example of using the network (i.e., plugging in an input face image into a placeholder, and getting the label as an output.)

2.5 Bonus: Gradient Visualization

Here, you will use Utku Ozbulak's PyTorch CNN Visualizations Library to visualize the important parts of the input image for a particular output class. In particular, just select a specific picture of an actor, and then using your trained network in Part 4, perform Gradient visualization with guided backpropagation to understand the prediction for that actor with respect to the input image. Comment on your results.

What to turn in

You will be turning in a written report of your solutions in a single PDF file. **These reports should be prepared using LaTeX using NeurIPS style** and will be submitted via e-mail. A zip file containing the corresponding template and style files are provided at the course website. **The write-up must be 6-8 pages long.** You should present your results in a clear way using well-designed tables, plots and figures. Do not include your codes.

Grading

The practical will be graded out of 100 points: 0 (no submission), 20 (an attempt at a solution), 40 (a partially correct solution), 60 (a mostly correct solution), 80 (a correct solution), 100 (a particularly creative or insightful solution). **Note that the grading depends on both the content and clarity of your report.**

Late Policy

You may use up to five *slip days* (in total) over the course of the semester for the three practicals you will take. Any additional unapproved late submission will be weighted by 0.5 and no submission after five days will be accepted.

Academic Integrity

All work on assignments must be done individually unless stated otherwise. You are encouraged to discuss with your classmates about the given practical, but these discussions should be carried out in an abstract way. That is, discussions related to a particular solution to a specific problem (either in actual code or in the pseudocode) will not be tolerated. In short, turning in someone else's work, in whole or in part, as your own will be considered as a violation of academic integrity. Please note that the former condition also holds for the material found on the web as everything on the web has been written by someone else.