EECS 553 HW 6

Due Thursday, October 24, by 11:59 PM Eastern Time via Gradescope

When you upload your solutions to Gradescope, please indicate which pages of your solution are associated with each problem. Also please note that your code should be uploaded to Gradescope (as a .pdf) and Canvas (as .py) as discussed below.

1. Neural Networks with PyTorch [5 points for each except for optional problems]

In this problem, we will explore supervised deep learning techniques for the problem of image classification. Specifically, we will be building and training a deep convolutional neural network (CNN) and exploring transfer learning as well as imbalanced datasets.

We will be utilizing PyTorch, a very popular deep learning package for Python that was primarily developed by Facebook's AI Research lab. This package has numerous features and handles complications during model training like computing gradients for you. To install this package, you can follow the instructions here: `https://pytorch.org/get-started/locally`. The code in this problem should run in a reasonable time on any CPUs, so you do not need to worry about using GPUs.

For part (a)-(f), we will train a CNN for classifying images of dogs into five different breeds: African hunting dogs, Chihuahuas, Dhole, Dingo and Japaneses Spaniel. The dataset we will be using contains around 800 JPG images of dogs of five different breeds. This dataset is provided to you on Canvas. Each image is of size $224 \times 224 \times 3$, where the 3 represents the color channels (red, green, and blue). The images have already been split into training set (around 75% images) and validation (testing) set (around 25% images). This dataset is balanced, so each class has a roughly equal number of training examples.

For your convenience, things you have to do throughout this problem will be **bolded**. In the code, sections you have to write code in will be prefaced with #TODO. Some details about the specific functions to use may be missing from the specifications; this is done on purpose. You will have to look up documentations to successfully complete this problem. Details about submission are available at the end of this problem.

(a) Data loading and preprocessing

Before we train our classifiers, we have to first load and preprocess our data. This step is very important, because real datasets contain a lot of noise that are not helpful during classification. Preprocessing the data allows us to highlight the true structure of the data. In this section, we will explore how to transform our data using PyTorch. We provide code for creating the data loaders and visualizing the data in `dataset.py`.

In PyTorch, you can specify a list of data transformations to be applied to the data. You can find the documentation for every available transform at `https://pytorch.org/docs/stable/torchvision/transforms.html`. **Complete the `get_transforms` function** in the `DogDataset` class when `if_resize=True` (i.e when resizing is needed) by filling in the list of PyTorch transforms according to the following list of operations:

- Resize the image to size $32 \times 32 \times 3$.
- Convert the datatype of the image to a PyTorch tensor.

- Normalize the image.

You may notice that there are two important arguments to the normalization operation, which are the per-channel means and standard deviations. **Complete the `compute_train_statistics` function** by computing these statistics on the training images. Use these statistics as input to the normalization operation and **report your computed statistics**. You can run `dataset.py` to see some training images after applying the data transformations.

Answer the following questions regarding data preprocessing:

1. What are the trade-offs of resizing and normalizing our images? List one benefit and one drawback for each technique.
2. Why should we only compute the per-channel mean and standard deviation on the training images and not on the validation images?

(b) Creating your network

Now that our data is prepared, we can build the actual CNN to classify our images. Recall the basic building block of a CNN: a convolutional layer, a non-linear activation function, and some form of pooling. These building blocks represent the feature encoder portion of the network, which encodes our input images to some learned representation space. Once we have this representation space, we use a sequence of fully connected layers with non-linear activation functions to learn a non-linear mapping between the feature encodings and the target classes. For both part (b) and part (c), we will be building an architecture presented in Appendix A.

Answer the following questions regarding model creation:

1. How many learnable parameters does our network have? Compute this by hand. You should show the number of parameters for each layer separately as well as the total number of parameters. *Hint:* You can use the `count_parameters` function to check your work after you complete part (c).
2. Why do we randomly initialize the parameters of our network? Would there be an issue with initializing every parameter as 0.0?

(c) Creating your network (Implementation)

We provide a basic skeleton of a PyTorch network in `model.py`. When defining a network in PyTorch, you have to subclass the `torch.nn.Module` class to let PyTorch know to handle gradient computations automatically. **Complete the following functions in `model.py`** to complete the definition of our model:

- `__init__()` should contain definitions of each layer in the network, which are used by `forward(x)`.
- `init_weights()` should initialize all the parameters of the network.
- `forward(x)` should compute the forward pass of a given batch of input images by passing them through each layer and activation function.

You should implement your network according to the architecture presented in Appendix A at the end of this document. A portion of the code have already been implemented for your reference. You can run `model.py` to test whether there are any size mismatches.

(d) Training your network

We can now train our network for our classification task. In `train.py`, we provide a basic skeleton of the training procedure and code for visualization. We also provide two

additional files `checkpoint.py` and `plot.py`. `checkpoint.py` helps us save and load our trained models. `plot.py` allows us to plot the losses and accuracies during training. You only need to modify `train.py`.

Fill in the definitions of the loss function and the optimizer according to the following specifications:

- Loss function: Cross-entropy loss
- Optimizer: SGD (besides learning rate and momentum, use default value for parameters)

Documentation for loss functions can be found at `https://pytorch.org/docs/stable/nn.html#loss-functions` and documentation for optimizers can be found at `https://pytorch.org/docs/stable/optim.html`.

Complete the functions `predictions` and `accuracy`. More detailed specifications can be found in the skeleton code.

Normally, you will need to tune the hyperparameters using holdout/cross-validation. To save time, we provide you with the batch size, learning rate, momentum paramter and number of training iterations. Run `train.py` to train our model (should take approximately 1 minute). Checkpoints for your model at every epoch (one iteration through all the training examples) will automatically be saved, and you can resume training from any checkpoint. A plot should automatically show up, which allows you to monitor the model's performance at every epoch. **Include the final saved plot** in your write-up and **report the final accuracies and losses** of your model.

Note: For students who are using jupyter notebook you would want to add a line `%matplotlib widget` on the top of your note book. The plot will still not update interactively, but it will show the correct results at the end of the training.

(e) Answer the following questions regarding model training:

1. What is the relationship between training loss and validation loss? What would happen to each loss if we continue training the model?
2. Based on the losses, when should we stop the training of our model? Explain why we should not try to maximize the training accuracy (or minimize the training loss).

(f) Transfer learning

In the previous subproblem, we initialized our CNN with random values and proceeded to train the network from scratch. In practice, people rarely do this, because they usually do not have access to large datasets and thus it is difficult to learn good representations. Instead, they use a technique called transfer learning: rather than training our model from scratch, we can pre-train our model on a very large dataset first to learn good representations and then train the model on our target dataset. We can use the pre-trained model either as an initialization for our model or as a fixed feature encoder. This bypasses the aforementioned issue and can lead to much better model performance.

A common dataset that is used for transfer learning is the ImageNet dataset, which contains 1.2 million images with 1000 categories. Training our model on this dataset would take a very long time due to its size. Luckily, PyTorch provides pre-trained weights for a variety of different architectures that are each trained on the ImageNet dataset. We can directly utilize these pre-trained weights by using the `torchvision.models` package (documentation at `https://pytorch.org/docs/stable/torchvision/models.html`). First, set up the dataset in `dataset.py`. **Complete the** `get_transforms` **function** in

the `DogDataset` in the case `if_resize=False` (i.e no resizing of the image is needed) by filling in the list of PyTorch transforms. This should only differ from part (a) by 1 line. Now, **complete the `load_pretrained` function** in `transfer.py` according to the following specifications:

- Load a ResNet-18 model from `torchvision.models` with pre-trained weights. This should automatically download the weights to your local directory. Note that the ResNet-18 model is much larger than our previous model (around 30x), so training will take longer.
- Freeze all the parameters of the model besides the final layer by setting the `requires_grad` flag for each parameter to False. This will speed up training since we do not need to compute the gradient for most parameters.
- Replace the final fully connected layer with another fully connected layer with `num_classes` many output units. We have to do this because the original final layer has 1000 output units for the 1000 ImageNet classes.

Run `transfer.py` to load the pre-trained model and fine-tune the final layer on our dataset (should take approximately 6-10 minutes). This file will use the training code from the previous subproblem. **Include the training plot** in your write-up. Due to better learned representation of images, you are expected to see a significant improvement in validation accuracy compared to part (d).

(g) Imbalanced dataset (Optional)

So far, we worked with datasets that are balanced, *i.e.* there is an equal proportion of training examples in each class. Unfortunately, many machine learning datasets in the real world are imbalanced, which introduces problems during training. In this subproblem, we will explore the difficulties caused by class imbalance and methods to deal with them.

For the last two subproblems, we will use a binary classification dataset, and the task is to classify whether the image is of a cat or a dog. However, the dataset is unbalanced: the dataset contains 200 training examples of cats but only 10 training examples of dogs, and 100 validation examples for cat but only 10 validation examples for dogs. The images have different dimensions, so we have to resize them to the same size.

First, set up the dataset in `dataset.py`. **Complete the `get_transforms` function** in the `DogCatDataset` by filling in the list of PyTorch transforms according to the following list of operations:

- Resize the image to size $256 \times 256 \times 3$.
- Crop a $224 \times 224 \times 3$ square from the center of the image.
- Convert the datatype of the image to a PyTorch tensor.
- Normalize the image with mean $= [0.485, 0.456, 0.406]$ and std $= [0.229, 0.224, 0.225]$.

Answer the following question:

- What is the issue of using accuracy when evaluating models under class imbalance?

(h) Imbalanced dataset cont'd (Optional)

To better see what issues this class imbalance will cause, we need to introduce several new metrics besides the standard accuracy. **Complete the code** for the following metrics in `imbalance.py`:

- Per-class accuracy - Instead of computing the accuracy over every image, compute the accuracy for the images of each class.

- Precision - Compute the proportion of positive label predictions that are correct (treat dog class as positive).
- Recall - Compute the proportion of images with positive labels that the model predicted correctly.
- F1-score - Compute the harmonic mean of precision and recall, which is f1-score $= 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$.

Run `imbalance.py` to evaluate the same model from subproblem (f) with the exceptions that the final output layer is modified for binary prediction on the imbalanced dataset with our new metrics. To save you some time, we provide the model checkpoint after training for 5 epochs. **Include the numbers** in your write-up and **answer the following question**:

- What is the issue with training on an imbalanced dataset according to the metrics?

Now that we know what the issue is, we can explore methods for addressing the class imbalance. Since one class has much less data, we can try to balance the dataset by duplicating the images in the dog class. Another simpler approach is to scale the loss associated with the dog class so it weighs the same as the cat class. We will take the latter approach. **Fill in the definition** of the loss function in `train.py` (the same cross entropy loss function), but this time use a weight vector to rescale the loss associated with the dog class by 20.0 (hint: see documentation for cross-entropy loss). Run `imbalance.py` again to train and evaluate your model with weighted cross-entropy loss (should take approximately 6-10 minutes). **Include the numbers** in your write-up and **compare the performance** of the two models in a few sentences.

Submission

Make sure everything we ask for (**bold** items) are included in your write-up. For each figure / result in your write-up, clearly indicate which problem it is for. Attach your code to your write-up (copy-and-paste or screenshot) and submit it to Gradescope. Additionally, submit your code as a zip file to Canvas. Only zip the following files: `dataset.py`, `imbalance.py`, `model.py`, `train.py`, and `transfer.py`.

Appendix A. CNN Architecture

Layer 1. Convolutional layer 1.

- Number of filters: 16
- Filter size: 5×5
- Stride size: 2×2
- Padding: 2
- Activation function: ReLU
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{2}{5 \times 5 \times 3}$
- Bias initialization: constant 0.0
- Output size: $16 \times 16 \times 16$

Layer 2. Convolutional layer 2.

- Number of filters: 32
- Filter size: 5×5
- Stride size: 2×2
- Padding: 2
- Activation function: ReLU
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{2}{5 \times 5 \times 16}$
- Bias initialization: constant 0.0
- Output size: $32 \times 8 \times 8$

Layer 3. Convolutional layer 3.

- Number of filters: 64
- Filter size: 5×5
- Stride size: 2×2
- Padding: 2
- Activation function: ReLU
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{2}{5 \times 5 \times 32}$
- Bias initialization: constant 0.0
- Output size: $64 \times 4 \times 4$

Layer 4. Convolutional layer 4.

- Number of filters: 128
- Filter size: 5×5
- Stride size: 2×2
- Padding: 2
- Activation function: ReLU
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{2}{5 \times 5 \times 64}$
- Bias initialization: constant 0.0
- Output size: $128 \times 2 \times 2$

Layer 5. Fully connected layer 1.

- Activation function: ReLU
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{1}{256}$

- Bias initialization: constant 0.0
- Output size: 64

Layer 6. Fully connected layer 2 (output layer).

- Activation function: none
- Weight initialization: normally distributed with $\mu = 0.0, \sigma^2 = \frac{1}{32}$
- Bias initialization: constant 0.0
- Output size: 5 (number of classes)

Appendix B. Useful Links

- Documentation: `https://pytorch.org/docs/stable/index.html`
- Tutorials: `https://pytorch.org/tutorials`

Appendix C. References

- Random initialization we used is based on section of 2.2 of `https://arxiv.org/pdf/1502.01852.pdf`
- The original paper for ResNet `https://arxiv.org/pdf/1512.03385.pdf`