CSE 253: Programming Assignment 3

Winter 2019

Instructions

Due Saturday, February 16^{th}:

1. **Start on this assignment NOW!** If you have any questions or uncertainties about the assignment instructions in Part 1 or Part 2, please ask about them as soon as possible (preferably on Piazza, so everybody can benefit). We want to minimize any possible confusion about this assignment and have tried very hard to make it understandable and easy for you to follow.

2. You will be using PyTorch (v 0.4.0), a Deep Learning library, for the tasks in this assignment. See the post on Piazza by Jenny as to how to access the UCSD GPU server, the data, and PyTorch. Additionally, any updates or modifications to the assignment will be pushed to the PA3-CNN repository on Github, though we will try making a point to announce these clearly on Piazza.

3. Please work in teams of 3-4 individuals (no more than 4). Under *very special circumstances, like you have the plague, and so will endanger any teammates lives,*, we will allow you to do it solo. Please discuss your circumstances with us first to get approval.

4. ***Please don't post your code on github, at least not publicly! And don't post code on piazza, obviously.***

5. Please submit in your assignment via Gradescope. The report should be in NIPS format or another "top" conference format (e.g. IEEE CVPR, ICML, ICLR) - we expect you to write at a high academic-level (to the best of your ability). Make sure your code is readable and well-documented - you may want to reuse it in the future.

Learning Objectives

1. Understand the basics of convolutional neural networks, including convolutional layer mechanics, max-pooling, and dimensions of layers.

2. Learn how to implement a CNN architecture in PyTorch for image/object classification using best practices.

3. Build intuition on the effects of modulating the design of a CNN by experimenting with your own (or "classic") architectures.

4. Learn how to address the imbalanced/rare class prediction problem in multiclass classification.

5. Visualize and interpret learned feature maps of a CNN.

6. Learn the intricacies of transfer learning and experiment with it

Part I

Individual Assignment: Understanding Convolutional Network Basics

This portion of the assignment is to build your intuition and understanding the basics of convolutional networks - namely how convolutional layers learn feature maps and how max pooling works - by manually simulating these. We highly recommend reading the Stanford page on convolutional networks. Especially scroll down and take a look at the animated **Convolution Demo**.

For questions 1-3, consider the $(5 \times 5 \times 2)$ input with values drawn from $\{-1, 0, 1\}$ and the corresponding $(3 \times 3 \times 2)$ filter shown below.

Input Features

-1	0	0	-1	1
0	1	1	-1	0
1	1	1	1	1
-1	0	-1	1	0
0	0	1	1	1

-1	-1	-1	0	-1
-1	0	-1	-1	0
0	-1	0	0	1
0	1	1	1	1
1	0	1	1	1

Filter 0

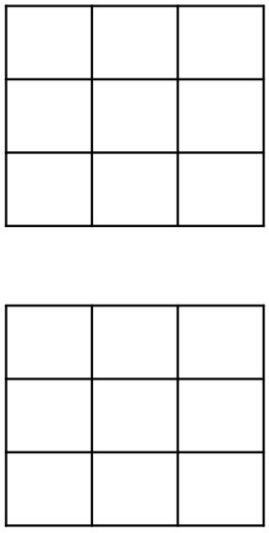
-1	-1	-1
0	0	0
1	1	1

1	0	-1
1	0	0
1	1	1

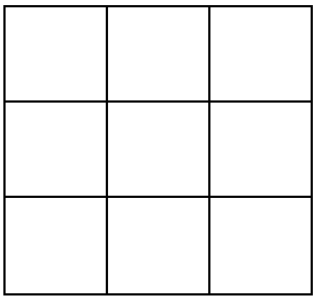
1. In the provided area below, fill in the values resulting from applying a convolutional layer to the input with no zero-padding and a stride of 1. Calculate these numbers before any activation function is applied. If there are any unused cells after completing the process in the provided tables, place an "X" in them.{5pts}

2. Think about what ideal 3×3 patch from each of the input channels would maximally activate the corresponding 3×3 filter. Fill in these maximally activating patches in the area below. Note that the numbers should come from the same set as before: $\{-1, 0, 1\}$, and use 0 where the element of the patch doesn't matter. (Hint: You will use *all* of these cells).

Maximally
Activating Patch



3. **Spatial pooling:** Using your **output feature map** in question 1, first apply ReLU, and then apply *max-pooling* using a $[2\times2]$ kernel with a stride of 1. Recall from lecture that spatial pooling gives a CNN invariance to small transformations in the input, and max-pooling is more widely used over sum or average pooling due to empirically better performance. Again, if there are any unused cells after completing the process in the provided tables, place an "X" in them.



4. **Number of learnable parameters**
Suppose we had the following architecture:

```
inputs -> conv1 -> conv2 -> conv3 -> maxpool -> fc1 -> fc2 (outputs)
```

where all convs have a stride of 1 and no zero-padding, and the inputs are a greyscale image (single channel). Conv1 has a kernel size of $[8 \times 8]$ with 12 output channels, conv2 has a $[8 \times 8]$ kernel with 10 output channels, conv3 has a $[6 \times 6]$ kernel with 8 output channels, and maxpool has a $[3 \times 3]$ kernel.
If the inputs are $[512 \times 512]$ greyscale images, what are (Hint: Don't forget the bias!):

(i) The number of parameters required for **conv1**: _____

(ii) The number of parameters required for **conv2**: _____

(iii) The number of parameters required for **conv3**: _____

(iv) Size of the effective receptive field of a neuron in **conv1**: _____

(v) Size of the effective receptive field of a neuron in **conv2**: _____

(vi) Size of the effective receptive field of a neuron in **conv3**: _____

(vii) What are the number of inputs to **fc1**? Show your work.

(viii) If fc1 is 100-dimensional and fc2 is 100 dimensional, how many parameters are there between the two layers (Again, include the bias!)? How does this compare to the total number of parameters in the convolutional layers?

(ix) Dropout is a very effective regularization technique in deep neural networks. It's only (meta-)parameter is a regularization term called the drop_probability which is the probability that a neuron will be dropped from the network. **Write 2-3 lines about what you understand about dropout and why you think dropout works. Feel free to read about it from it's original paper[2] or on the web, but put this in your own words!**

Part II

Group Assignment: Deep Convolutional Network for Thorax Disease Detection

Problem statement: Deep convolutional neural networks have been applied to a broad range of problems and tasks within Computer Vision. In this part of the assignment, we will explore a classification task which is increasingly relevant and important in the medical community - disease detection. We will build a deep CNN architecture and train it from scratch to predict thoracic disease(s) present in chest X-rays.

The **ChestX-ray14**[1] dataset contains 112,120 images (frontal-view X-rays) from 30,805 unique patients, where each image may be labeled with a single disease or multiple diseases. As such, we must design our CNNs for multi-label classification - clearly, applying softmax at the output layer will not achieve this and is therefore not an appropriate choice in this task. Additionally, the disease occurrences are not equally frequent - some are rare, some are common - so using basic classification accuracy will not be a sufficient metric in evaluating your model's performance. This situation is called a *class imbalance*. In fact, a baseline is to just always give the most frequent disease. Compute what this error rate would be and put it in your report.
Please refer to the dataset README from NIH for more details about the dataset and problem setting.

Implementation Instructions

We have provided you with a data loader, **xray_dataloader.py**, **ChestXrayDataset**: a custom PyTorch Dataset class, specifically designed for the *ChestX-ray14* dataset, and a function **create_split_loaders()**, which partitions the dataset into train, validation, and test splits while creating PyTorch DataLoader object for each. The latter is an iterator over the elements in each split of the dataset which you will use to retrieve mini-batches of samples and their labels. Please familiarize yourself with the code and read the comments.

1. **Evaluating your model:**

   Before we discuss the neural network details, we must clarify how you will accurately and transparently evaluate your model's performance. When dealing with class imbalance, there are cases - particularly in disease and anomaly detection - where the naive prediction will result in very high overall accuracy (e.g. predicting "no disease"). We will compute accuracy, but we will also use the notions of *Precision* and *Recall*. These are computed using three things: the *True Positives (TP)*, the *False Positives (FP)*, and the *False Negatives (FN)* (also called *Misses*). TP is the number of diseases present that your model says are present, FP is the number of diseases you say are present that aren't, and FN (or misses) is just what it sounds like: the number of times you did not report that a disease is present when it is. Look up Precision and Recall on Wikipedia for an explanation. You will be reporting the following on a **per-class basis**:

   (i) Accuracy: percent correct predictions $= \frac{\text{total correct predictions}}{\text{total number of samples}}$

   (ii) Precision: $\frac{|TP|}{|FP|+|TP|}$.

   (iii) Recall: $\frac{|TP|}{|TP|+|FN|}$.

   (iv) Balanced classification rate (BCR): $\frac{Precision+Recall}{2}$

   In addition, you will report:

   (v) The aggregated scores for precision, recall, and BCR, by computing an equally weighted average across all individual scores, respectively.

   (vi) A confusion matrix showing the classifications for all classes vs. all classes (demonstrating what your model "confused" as a different class). A confusion matrix, for our purposes, will have the actual diseases along the top, and the reported diseases along the side, so each entry is the fraction of times that your model reported X when the disease was X (the diagonal), as well as the number of times you reported the disease was X when it was really Y. For this purpose, add a row and column for "no disease present". So, for example, if your model predicted one disease when there are two, you would get an entry in the (no disease, disease2) cell, where the first entry is the row, and the second entry is the column. These

should be percentages. If you reported A when the diseases were B and C, you get an entry in the (A,B) and the (A,C) cell. The ideal matrix is diagonal, with 100% down the diagonal,

2. **Create a baseline model:**

   You will create a simple convolutional neural net for the purposes of (i) getting acquainted with PyTorch, (ii) getting baseline results to compare more complex architectures with, and approaches to solving this class imbalance problem. The baseline architecture is the following:

   ```
   inputs -> conv1 -> conv2 -> conv3 -> maxpool -> fc1 -> fc2 (outputs)
   ```

   Look familiar? Use the metaparameters described in Part 1, question 4. Using the starter code provided in **baseline_cnn.py**, complete the implementation for the architecture as described in the comments, replacing the instances of __ with its corresponding missing value. This includes finishing the **__init__()** and **forward()** functions.

   To run the **baseline_cnn.py** model, we have additionally provided a basic Jupyter notebook, **train_model.ipynb**, containing some starter code needed to train the model. Based on your reasoning for determining the activation function for the output layer of the network, determine which loss criterion you should be optimizing - this may be something you are already familiar with (note that PyTorch loss criteria can be found in the **torch.nn** package). Additionally, use ReLU activations, Xavier initialization of the weights, and use the Adam gradient descent optimizer, which can be found in the **torch.optim** package. You will need to pick an initial learning rate. Train the baseline model until the loss stops decreasing on the validation set.

   Please note that this is a very bare-bones implementation which may perform decently on MNIST but die a painful death when faced with a much more challenging problem such as this. As such, this architecture may have "good" overall classification accuracy, but fail to address the class-imbalance problem.

   In addition, make note that the data loader applies any specified color-scale conversion and transformations to your images *on-line*. In the next section, you are welcome to optimize this as you see fit (including preprocessing the complete dataset and saving a copy in your **home directory in the Kubernetes environment**.

3. **Experiment with variations of the baseline model (don't do these things to your baseline model above - we want it to do a bad job!):**

   To get a better sense of how your design choices affect model performance, we encourage you to experiment with various approaches to solving this multiclass classification problem using a deep CNN. You are welcome to make a copy of the **baseline_cnn.py** file and **train_model** notebook for this purpose (as well as convert the training code into a .py file). The following things can all be done at once - we don't mean for you to try one at a time. That would take forever! At a minimum, you should do the following:

   i. **z-score the images** on a per-image basis (so each image has 0 mean and unit standard deviation).

   ii. **Augment your dataset** by applying **at least 2** transformations to the input images: this can include using cropping, etc. Note that the original images are $[1024 \times 1024]$ dimensional, and in the **baseline_cnn.py** implementation, we resize them down to $[512 \times 512]$ to improve the speed of computation. You can try downsizing them further, but bear in mind that you may lose meaningful information at the cost of improved efficiency. However, this would allow you to experiment more quickly, and hopefully improvements would transfer up to the $[512 \times 512]$ images. **Use the same transformations across all architectures (except the baseline model.)**

   iii. **Try two distinct, additional architectures:** This includes making significant changes in the number of layers, activation functions, dimensions of the convolutional filters, etc. Your own architecture should be significantly different from the baseline architecture we provided. Simply adding an additional layer is not sufficient!

   iv. **Address the rare class or imbalanced class problem.** This is often done by **enforcing** that the network learn to categorize the infrequently seen classes. You can use: (1) weighted loss, (2) using batches of evenly distributed class representation (resampling). At a minimum, you must implement your own weighted/balanced loss criterion. Report the performance of your method.

v. **Experiment with Dropout** Experiment with different dropout rates for your best model.

vi. **Ensembling:** When you have two good models, try combining them with the baseline in a voting scheme. Or just average the outputs of the two models to see if that improves your results (two heads are better than one...).

4. **Experiment with Transfer Learning** In the previous sections, we trained our model from scratch (randomly initializing weights). But in this section, we will transfer weights from a pre-trained network. You have 2 tasks: 1) use a network pre-trained **on Imagenet only**: freeze all the layers and remove the last layer, then stack your own last layer and train it on our dataset. 2) Try fine-tuning: By this we mean, use a pre-trained network **again on Imagenet only**, and do not freeze the weights - start with the pretrained weights, and then fine-tune them with backprop (along with your own last layer, of course). Report the performance of each.

For each of the above models, you are expected to provide the loss curves for the train and validation sets, and evaluate them as you did the baseline model in Part II.1.

What to include in your report

In addition to learning very useful and applicable skills in deep learning and computer vision, this portion of the assignment will help you learn to write a scientific report. Please include an abstract and the following *7 sections*:

0. **Abstract**{5 pts.}
   The **Abstract** should serve as a one paragraph synopsis of the work in your report, including the task (e.g. disease tagging on dataset X), how you approached it, and a quick overview of your results.

1. **Introduction**{7 pts.}
   The **Introduction** should describe the problem statement or task, why it's important, and any necessary background your audience may need to know. Keep in mind that since we're using Xavier weight initialization and batch normalization (see starter code), you should understand the basic mathematics behind these concepts at a minimum, how they affect your network parameters, and why they're useful. As such, these should be discussed in either the **Introduction** or **Methods** sections.

2. **Related Work**{5 pts.}
   The **Related Work** section should review any work you used to inspire your approach - this includes previous research in the specific problem you address, as well as any model architectures/core ideas you build upon. You should include citations in standard reference format with this itemized in a **References** section at the end of the report. This is where using LaTeX and BibTex can come in very handy.

3. **Methods**{20 pts.}
   In the **Methods** section, you should describe the implementation and architectural details of your system - in particular, this addresses how you approached the problem and the design of your solution. For those who believe in reproducible science, this should be fine-grained enough such that somebody could implement your model/algorithm and reproduce the results you claim to achieve.

   (i) **Baseline:** You should describe the baseline architecture, stating the appropriate activation function on the last layer of the network, and the loss criterion you optimized.

   (ii) **Experimentation:** Encapsulate your two experimental CNN architectures, each in a 2-column table where the first column indicates the particular layer of your network and the second column states the layer's dimensions (e.g. in-channels, out-channels, kernel-size, padding/stride) and activation function/non-linearity.

4. **Results:** {20 pts.}
   In the **Results** section, you should demonstrate your models' performance compared with the baseline implementation. You should include all the performance metrics described in Implementation Instructions part 1 for **each implementation based on your test set results**. Please organize these results into a series of concise tables. The formatting is your choice, so long as it is easily interpretable.

   Additionally, you should include the following for each architecture:

(i) A single plot showing both training and validation *loss* curves

(ii) A table including the performance metrics outlined in "Evaluating your model" (Implementation Instructions Q1).

(iii) Visualizations of the learned filter maps **from your best performing network only**. Select 1 filters from an early layer, 1 filters from a middle layer, and 2 filters from a later layer.

5. **Discussion:** {25 pts.}
   The Discussion section should highlight the meaning of your results and the approaches followed in this work. At a minimum, please discuss the following important points, but feel free to go beyond this:

   (i) How did the performance of your implementations differ; hypothesize why this was the case.

   (ii) What common confusions between classes did your model exhibit; why might this have occurred? Did this differ significantly between your different architectures?

   (iii) What do the different performance metrics tell us about the 5 models' abilities to identify diseases? What information does precision and recall offer about the decision boundary learned by each model us as opposed to accuracy or our plotted loss?

   (iv) Discuss the visualizations of the feature maps and how they vary with depth.

6. **Authors' Contributions and References**
   Each group member must write a separate paragraph describing their contributions to the project.

   Don't forget include the references you cite!

References

[1] X. Wang, Y. Peng , L. Lu Hospital-scale Chest X-ray Database and Benchmarks on Weakly-Supervised Classification and Localization of Common Thorax Diseases. Department of Radiology and Imaging Sciences, September 2017. https://arxiv.org/pdf/1705.02315.pdf
[2] N. Srivastava, G. Hinton, A. Krizhevsky Dropout: A Simple Way to Prevent Neural Networks from Overfitting https://www.cs.toronto.edu/ hinton/absps/JMLRdropout.pdf