CSCI1420 Capstone Report

Kristen Cai

May 16, 2025

1 Background

1.1 Task: Sheet Music Optical Recognition

I have taken MUSC0680: Chamber Music Performance for two semesters during my time at Brown. One challenge I have encountered as a pianist is accessing high quality sheet music. Each piece often has editions from different publishers. Between publishers, there is a large variety in the amount of detail given for suggested fingerings, how measures are counted, how much dynamic detail is suggested, etc. Many editions of music that have more of these useful details are older physical copies that may be discolored or hard to read once scanned. For music available in the public domain, there is in most cases a trade-off between ease of reading music and the details it provides, which can be a frustrating experience for musicians looking for the best sheet music to use.

One solution to this is sheet music Optical Recognition (OMR). This type of technology reads in scanned sheet music and converts it digitally into a cleaner format. In the process, it makes the music easier to read by removing discoloration or improper alignments that occur in the process of scanning, while preserving the details included in the edition. Especially in today's age, where many people read music digitally rather than from physical copies, this is very useful for tablets and other music reading applications.

The first step of this task, which my model focuses on, is recognizing musical symbols. Musical symbols vary in size and variety. Larger symbols include musical staves (the entire staff line that the notes are written on), medium symbols may include clefs or dynamic markings, and smaller symbols include note heads, accidentals, and rests. Because of the nature of sheet music, many symbols may overlap or be very close together. These symbols often have different variations, (note heads can be filled in or not, rests have many different variations, etc.) increasing the difficulty of the task.

1.2 Faster R-CNNs

Many object detection and image segmentation tasks use deep learning methods. Since sheet music OMR requires classification of small, potentially overlapping symbols, it requires a method with high precision. For this task, I chose to use Convolutional Neural Networks (CNNs), specifically region-based CNNs (R-CNNs). The variant I chose to use, Faster R-CNN, detects objects in two stages:

- The first stage finds Regions of Interest (RoIs) that may contain objects (which is faster than other methods used, hence the name of the CNN).

- The second stage takes the RoIs, classifies the object in the region, and updates the coordinates of the bounding box.

The backbone of the Faster R-CNN is a regular CNN used to extract features.
I also chose a Faster R-CNN with a Feature Pyramid Network (FPN), which creates more feature maps, allowing it to suggest RoIs of different scales. For sheet music OMR specifically, there is a large variation in size of symbol, so this allows the backbone to detect both small overlapping symbols and larger symbols with higher accuracy.

2 Approach

2.1 Data Preprocessing

I trained the model using the DeepScores V2 dataset, which has two versions. While one of them is much larger, I used the smaller dense dataset since I was limited by computation power. This dense dataset includes 1,714 diverse score images from MuseScore. Each image in the dataset has corresponding annotations that label specific locations (bounding boxes) and categories of musical symbols present in the image, as well as the symbol category. There are around 350 different symbols across categories, including variants of notes, clefs, accidentals, rests, etc.
The dataset provides its annotations in a custom JSON format, which I converted into the Common Objects in Context (COCO) format (see `convert.py`, a standard widely used for object detection tasks. Each COCO image must include metadata including the image's ID, filename, width, and height. I extracted information about each image from the original JSON to match this structure. For each annotated object, I converted its bounding box to COCO's required format: `[x, y, width, height]`.
Each musical symbol category in the dataset (e.g., `"clefCTenor"`, `"fermataBelow"`) was also mapped to a unique numerical ID. After preprocessing, all image metadata, annotations, and category mappings were compiled into a single dictionary and saved as a COCO-style JSON file (see `deepscores_train_coco.json` and `deepscores_test_coco.json`).

2.2 Data Transformation

In my main code file, `main.py`, the DeepScoresDataset class reads in the preprocessed COCO JSON annotation file and builds a map from the original category IDs to an increasing set of labels starting at one. To simulate common sheet music scanning issues, each score image is randomly rotated by up to 10 degrees (simulating scanning misalignments) and its brightness and contrast are altered slightly (simulating variations in lighting). Then the image is resized to 512x512 pixels to increase the visibility of small symbols. Then it is converted to a tensor and normalized using ImageNet. Finally, the original bounding box coordinates are rescaled to match the new image dimensions after resizing.

2.3 Model Architecture

The core of the model is PyTorch's Faster R-CNN model with a MobileNetV3-Large FPN backbone (see above for an explanation). I started with a model pretrained on COCO, then replace its final classifier with a new head, FastRCNNPredictor, that knows one "background" class (for a region that does not contain any musical symbols) and one for each symbol type. To speed up training and avoid over-fitting, I only trained the last 20 layers.

2.4 Model Training

This model used a standard 80/20 training/validation split and a stochastic gradient optimizer. The built-in loss function uses a combination of cross-entropy loss and Smooth L1 loss (often used with bounding boxes).
The final accuracy was measured using average precision (AP) and average recall (AR). AP@[0.50:0.95] (large) measures the average precision over Intersection over Union (IoU) thresholds (overlap between predicted and true bounding boxes) from 0.50 to 0.95, but only for objects classified as 'large' (a COCO-defined large threshold). It quantifies how many objects my model found are correctly labeled. Similarly, AR@[0.50:0.95] is the average recall computed across the same IoU thresholds (0.50 to 0.95) across all object sizes. It quantifies the proportion of musical symbols that my model found.
I started training my model at a baseline of 10 epochs without COCO-pretrained weights. This baseline model started with very high initial losses (Epoch 1/10 had a train loss of 18.2973 and a validation loss of 15.7374) that decreased only modestly by Epoch 10 (which had a train loss of 12.9603 and a validation loss of 13.5299). The model achieved a final AP and AR of 0.000, suggesting problems with my initial setup.
At this point, I mapped each category ID to a label starting at 1, rescaled my bounding boxes based on the scaling of my original images, and added ImageNet normalization (to match my model's MobileNetV3-Large FPN backbone, which was pretrained with this normalization). The first two steps decreased my training and validation losses greatly (by the 10th epoch, they were 2.1959 and 2.2201 respectively), though AP and AR did not change. Once I added the normalization, I had similar losses (2.1002 and 2.1089 for training and validation respectively) but AP and AR increased to 0.030 and 0.045 respectively.
Finally, switching to COCO-pretrained weights and extending training to 20 epochs produced a dramatic drop in both training and validation losses. By Epoch 1/20, the losses fell to 2.7069/2.2894, and by Epoch 20 the losses stabilized around 1.6806 (train) and 1.7530 (validation), yielding an AP@[0.50:0.95] of 0.058 and an AR@[0.50:0.95] of 0.077.
Finally, further increasing training to 40 epochs saw continued but incremental improvements. The final epoch (40/40) recorded losses of 1.5548 (train) and 1.6518 (validation), and the AP increased to 0.090.

2.5 Challenges

While developing and training my model, there were many challenges I encountered. Some of the approaches I took were effective, but some were not.

- Most object detection datasets are provided in a COCO format, but DeepScores V2 had its own custom format. I had to parse the given JSON into the correct COCO format.
- When training my model, I tried using a RetinaNet base model instead of FasterRCNN. RetinaNet is better at classifying rare symbols, which seemed to be a good for my smaller, diverse dataset. However, this model took significantly longer to run and did not lead to higher precision.
- After increasing the number of epochs to 40, I saw that the loss still appeared to be decreasing. I tried raising it to 60 and the loss decreased a little bit more, but the average precision did not.
- I tried to experiment further with changing the learning rate. I found that it took many epochs for the loss to decrease, and it decreased at a very slow rate. However, increasing

the learning rate to 0.01 led to exploding gradients as training started. I tried modestly increasing the learning rate to 0.007, which resulted in the loss decreasing at a faster rate, but no improvement in accuracy.

- In general, once I was able to have my model begin to create more accurate bounding boxes, it did a lot better of a job detecting larger symbols, but failed to recognize much smaller symbols. I tried resizing my images to have more pixels, which did not have an effect on precision either.

3 Results

Here is a table summarizing the results across the different configurations of models I described above.

Configuration	Train Loss	Val Loss	AP@[0.50:0.95] (large)	AR@[0.50:0.95]
Baseline	12.9603	13.5299	0.000	0.000
Mapping Labels / Resizing	2.1959	2.2201	0.000	0.000
ImageNet Normalization	2.1002	2.1089	0.030	0.045
COCO-pretrained, 20 epochs	1.6806	1.7530	0.058	0.077
COCO-pretrained, 40 epochs	1.5548	1.6518	0.090	0.102

Table 1: Training and validation losses, AP@[0.50:0.95] for large symbols, and AR@[0.50:0.95] across configurations.

The final results showed a relatively low precision (10%). This can be attributed to the difficulty to learning extremely small symbols for a task like sheet music OMR, as well as imbalances in the dense version of the DeepScores V2 dataset (as I was constrained by memory and computational resources).
However, visualizations of the bounding boxes that my model produced are more telling as to the types of predictions it was able to make.

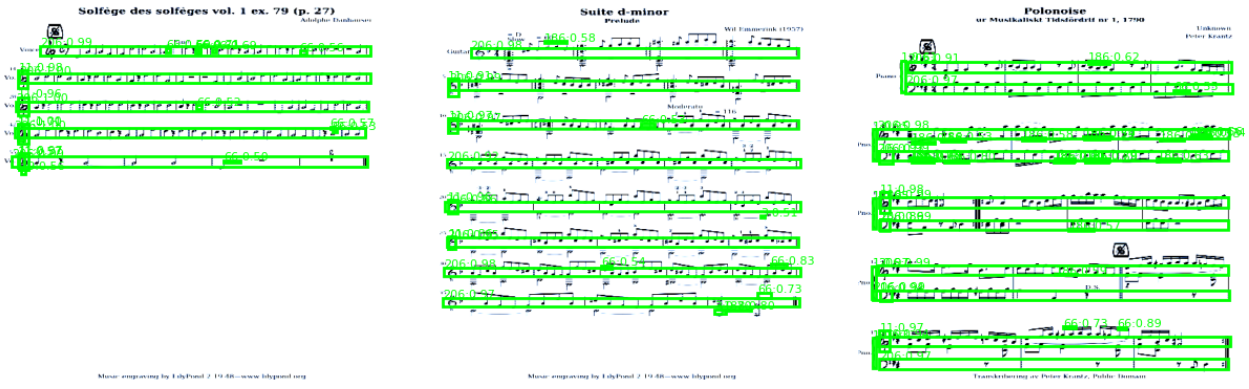

Figure 1: Bounding box predictions

From the visualizations of the predictions, it appears that the model was able to detect larger regions but had trouble isolating smaller musical symbols. Most of the boxes span whole staves (and sometimes individual measures) as well as different lines in the treble versus bass clefs, but miss individual notes and accidentals. Notes that diverge from the main staff are not caught. This

suggests that the model is on the right track in detecting symbols on a high level, but is not yet well trained on these small symbols.

4 Next Steps

If I were to continue working on this project, the main priority would be refining the model's ability to locate and classify small musical symbols. To accomplish this, I would introduce smaller anchor box sizes that more closely match the dimensions of note heads and articulation marks. I would also continue to experiment with resizing the images to have larger pixel dimensions to help the model better see the smaller symbols.
I would simultaneously continue to experiment with the most optimal hyperparameters by varying batch size, momentum, and weight decay to improve accuracy.
Another priority would be to addressing the class imbalance. Given more time, I would further explore using RetinaNet, which is better at classifying rare symbol classes.
Finally, since training often took many epochs, I would implement experiment with learning-rate scheduling. I believe that this model could benefit from a higher learning rate, but since I ran into exploding gradients when testing this, I would try using a smaller learning rate to start, increasing it to speed up learning, and decreasing it at the end for additional fine-tuning.

Overall, I learned a lot doing this project in applying concepts learned in class to a task I find applicable and useful in my own interests.