Scoring Images by Aesthetic Value

Neha Nishikant
nnishika@andrew.cmu.edu

Disha Das
dishad@andrew.cmu.edu

Abstract

Our goal in this project was to create a model that takes in an image and predicts a "score" evaluating how aesthetically pleasing the image is. This is useful because in many scenarios (Airbnb, Yelp, Poshmark, eBay) people upload images of a product in order to advertise. These images often vary drastically in quality. How appealing the image is in terms of quality, brightness, color, organization, etc. is important in getting people to purchase their products. This tool would allow the seller to upload their image and get an assessment of how appealing the image is and in a way predict how successful it will be in attracting customers. We used a VGG-inspired Double Column CNN architecture along with pretrained ImageNet weights as our model. Our results showed that the model was able to learn from the training data, but overfit, and thus did not generalize well to the test data set.

1 Introduction

1.1 Goal

Not all images are made equal. Some images are more aesthetically pleasing than others. Our goal is to train a model that can differentiate between the two. We aim to score an image based on its aesthetic value.

The applications of this are very wide and have marketable value. We narrowed our focus to just images of food and drinks. The value we aim to provide is for small businesses and restaurants to curate their media presence. We want to help restaurants put their best foot forward in terms of the images of their food that they advertise. Ideally, they could use this trained model to pick out the best photos they can use on their websites, posts, and general brand.

1.2 Dataset

Given a model that can successfully differentiate between pleasing and non-pleasing food and drink images, the model can be similarly trained on other classes of images to provide a similar value. The dataset we used is the Aesthetics Visual Analysis (AVA) dataset hosted on `https://github.com/mtobeiyf/ava_downloader`. The data is from `https://www.dpchallenge.com/image.php?IMAGE_ID=359334` which is a photo contest from 2006. People individually scored each image on a scale of 0 to 10.

In it's raw form, the label for each image were formatted similar to a frequency table for each score 0-10. We define a metric *score* for each image which is an average of all the individual scores that people submitted.

Many of the image labels in the training and test data set were concentrated in the range from $4 - 6$. We believe a data set that either has a more evenly distributed set of scores (representing the quality of the image) or had images that were clearly distinguishable as high or low quality may have yielded better results.

10417 Final Project (2020).

1.3 Findings Overview

We tried many different approaches and architectures which we will outline in the following sections. However, we saw a common pattern of over-fitting. Often the model would learn the train data, but test data did only slightly better than a naive estimate. We explore possible reasons for this in future sections.

2 Background

2.1 Preliminary Architecture Implementation Details

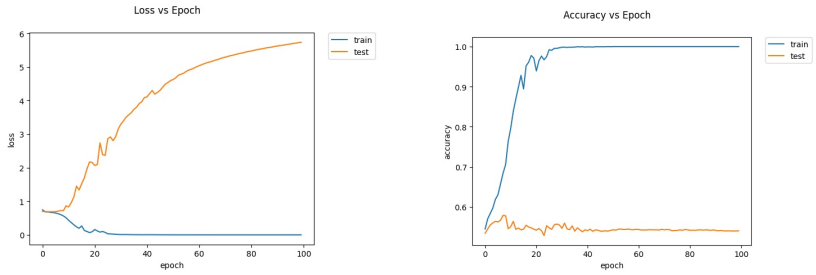
As discussed in the midway review, our original architecture was 3 convolution layers, each followed by a max-pool with ReLU activation, and a linear layer with 2 output nodes.

Our input data was a 3-channel (RGB) 300x300 matrix of the pixel values of an image. The dataset has a lot of hetereogeneous images with varying pixel sizes. To accomplish a 300x300 size, we either cropped or 0-pixel padded the top left of an image. We also further preprocessed the data to discard grayscale images and images below 100x100 pixels to avoid images that would need too much padding.

Our labels were a binary classification of 0 or 1. We preprocessed the data to obtain the average score for the train images. We then labeled each image with a 0 if its score was below the average and labeled it with a 1 if it was above.

2.2 Results

This massively overfit.



(a) Loss (b) Accuracy

Figure 1: Preliminary results

As seen in Figure 1, the train loss and test loss started in the same area. Train loss went down, but test loss actually increased. Further, the train accuracy plateaued at 1.0 very quickly. The fact that it even got to 100% accuracy is not a good sign, and these results taken together show the overfitting.

There reasons for this overfitting are

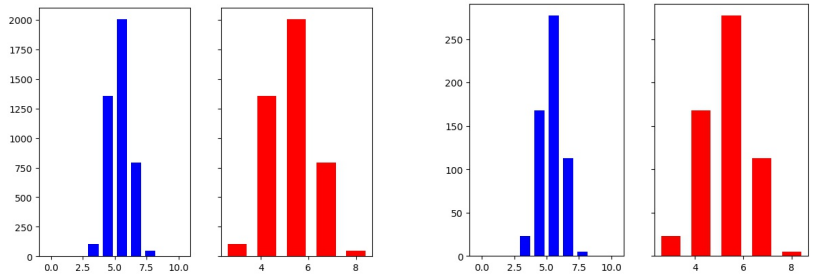
- Shallow architecture
- Easily memorizable labels
- Seemingly arbitrary labels

2.2.1 Architecture analysis

Our shallow architecture was a huge culprit here. The shallow, wide architecture essentially memorized all the labels and functioned as a look-up table for the train data. Thus it basically randomly guessed for the test data. We learned that shallow architectures do not generalize well.

2.2.2 Label analysis

Secondly, our train set is relatively small and since the labels are just 0 or 1, there aren't that many possible assignments of labels to images. This classification method essentially allows for $O(2^N)$ possible outputs/assignments. On the other hand, using regression would allow for a continuous output and thus it is far less memorizable since the number of assignments can't be captured in just $O(2^N)$. The memorizing factor contributed heavily to overfitting since the model can just memorize the train set without actually learning anything.



(a) Train (red is zoomed in) (b) Test (red is zoomed in)

Figure 2: Score distribution

Thirdly, the labels were given seemingly arbitrarily in the sense that there was no true underlying distribution that we were learning. The reason we say this is because the variance of the score data was very low for both train and test, therefore most of the images were scored very close to the mean. This can be seen in Figure 2.

Thus there wasn't actually anything the model could have truly learned and thus is resorted to memorizing.

3 Related Work

We mainly referenced "Rating Image Aesthetics using Deep Learning", which was the source of inspiration for using a Double Column CNN, which will be explained in further sections. In this paper, they experimented with a Single Column Neural Network (SCNN) as well as a Double Column Neural Network (DCNN) [2]. The SCNN tried four types of input image transformations before passing them through four convolutional and max-pooling layers followed by two fully connected layers. The image transformations included center crop, image-warp, or "anisotropically resizing", padding and random crop. Each image was given a label of 0 (low quality image) or 1 (high quality image). The paper found that the DCNN was able to correctly classify many images that the SCNN misclassified. We also used "Image Aesthetics Quantification with a Convolutional Neural Network (CNN)" for background and learn about ideas such as transfer learning [1]. Another paper builds off the DCNN approach but also factored in the content of the image, called "Scene Aware Information" [4]. This was done by training using semantic tags of what the images contained. They used the DCNN model to extract features and then used a Support Vector Machine with an RBF kernel as their classifier. This paper achieved the highest accuracy with a ResNet-50 model that uses local views, global views, and scene aware information.

4 Methods

4.1 VGG (inspired) Architecture

The first thing we tackled was the shallow architecture. Our second set of of model iterations followed the VGG-style architecture. We wanted to find a balance between accuracy and necessary computation. The VGG-16 architecture as a whole was too large to train for our purposes. We modified the architecture so that it would be more efficient to train. The architecture we used had three blocks of three 3×3 convolutions followed by a 2×2 maxpool. The first block had 16 output channels, the second block had 32, and the third block had 64. We then finished with a one fully

connected layer with ReLU activation going to 10 hidden units and a second fully connected layer with 1 output node.

Another aspect we changed was the input and output data. While it's difficult to map what exactly the effects of padding the images are, we can deduce that extra 0s along the top left can only hurt the model's predictions because of the heterogeneity of the inputs data. Thus instead we resized our images to be 128×128 pixels. This got rid of any need for padding.

For the output data, instead of sticking to classification, we switched to regression. As discussed above, regression is better to combat overfitting as the model needs to learn a continuous, more complex function. This remedied both the memorizable nature of the labels as well as the arbitrary classification of them around the mean. Using regression we haven't lost any information in our labels by reducing it to a 0 or 1. We used MSE loss for optimization.

These results, unfortunately, had these two main issues

- Low variance in predictions
- Still overfitting

4.2 Image Pairs

Since one of the issues with our dataset was that the images were all very close to each other in quality, we tried adjusting the problem we were trying to solve by seeing if our model could choose between two pictures and determine which has the better image quality. This involved preprocessing our data into image pairs where one image was of "significantly" higher quality than the other. We define significantly as images that have difference of at least 2 points in score.

The motivation behind this was reminiscent of how a human would solve this task and was meant to give the model more information. Calculating a score for an image is difficult even for a human and because it is so subjective, two people might not agree on a score. However, differentiating between images that had a significant difference in votes is easier for a human to learn. We experimented with the idea that this could help a model learn as well.

However, we found that the overfitting was worse, emphasizing the regression argument discussed in section 2.2.2. However, we didn't give up on pairs of images just yet. The following section outlines another way we used input pairs.

4.3 Double Column CNN Architecture

One successful approach in the literature was to use a Double Column CNN (DCNN). The motivation behind this is because we are rescaling all our images to be 128×128 pixels which can result in image distortion and information loss. Thus, the model has two inputs/two columns: a global view of the image and a local view of the image. The global view takes the full image and resizes it to be 128×128 whereas for the local view we randomly cropped a 128×128 section of the image. Since the model parameters are jointly trained, both the local and global views of the image are considered while training, which reduces information loss and increases the flexibility of our model. In the paper, the two columns were independent in the initial convolution and then eventually concatenated together as input into fully connected layers. Then the error was back-propagated in each column using stochastic gradient descent.

4.3.1 Architecture Implementation

We implemented this with the VGG inspired architecture outlined from section 4.1. Our input was a pair of images, the first being a random 128×128 pixel crop of the image, and second being entire image resized to 128×128. We ran these through our convolution layers. We then concatenated the output into and ran in through both fully connected layers.

4.4 Pretrained Weights

We observed in the literature that most approaches used transfer learning by starting with pretrained weights. We used the VGG-16 architecture from PyTorch which was trained on the ImageNet

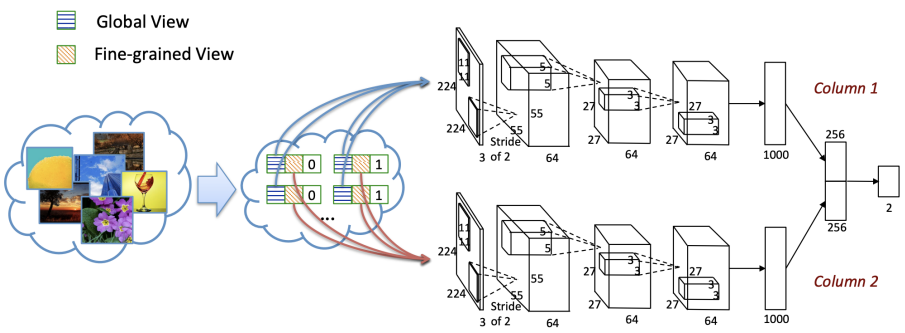


Figure 3: DCNN architecture [2]

database. The motivation behind this is that ImageNet is a huge database and the pretrained models are trained on thousands of images and 1000 output classes. We don't have this computational power, but we can use the pretrained model as a feature extractor to obtain a semantic meaning from our images that can be learned on.

4.4.1 Architecture Implementation

We combined transfer learning with our DCNN architecture. We ran each pair of images (random crop and resized) through the pretrained VGG-16. We then concatenated the results into a 2000×1 vector and ran it through additional fully connected layers that ended with 1 output node.

We used the VGG-16 weights as a feature extractor, so we did not fine tune or update these pretrained weights. Our optimization consisted of running MSE loss and optimizing only the added fully connected layers after the pretrained VGG and concatenation.

Further, to be compatible with the VGG-16 weights, we resized and cropped our images to be 224×244 instead of 128. We also normalized the input data.

4.5 General Modifications

4.5.1 Deeper Fully Connected Layers

The fully connected layers are the meat of our learning. Thus to combat overfitting, we added more fully connected layers so that our model would be able to actually learn the underlying distribution. Our final configuration consists of three fully connected layers with ReLU activation. The first has 128 hidden nodes, the second has 10, and we end with an output of 1.

4.5.2 Batch Normalization

Firstly, the model was predicting very similar results for every single input. This was due to the low variance of the input data as shown in Figure 2.

Further, our inputs data wasn't normalized. These in combination contributed to the low variance in predictions. To combat this, we added batch normalization before every fully connected layer. This improved the variance in predictions greatly.

4.5.3 Dropout

We experimented with different configurations of dropout to relieve overfitting. We found that increasing the dropout too much resulted in very unstable learning curves. We also found that having dropout too close to the final fully connected layer, hurt the model too much. Our final configuration was a 0.2 dropout before the first fully connected layer.

4.5.4 Increasing trainset

Another idea we considered was that the model did not have enough images to train on, which can contribute to overfitting, so we increased the training set by shifting some of the test images to train set. The dataset was presplit into 2449 train images and 2441 test images. We resplit this into 4305 train images and 586 test images.

5 Results

Since our baseline method ended up with 100% training accuracy and very low test accuracy, we knew we had a severe overfitting problem. We fixed this by changing our model from a shallow 2-layer model to a VGG with deeper fully connected layers. After this adjustment, we were no longer overfitting, but neither our training nor test curves showed that they were learning anything at all, as they stayed about constant throughout which can be seen in Figure 4.

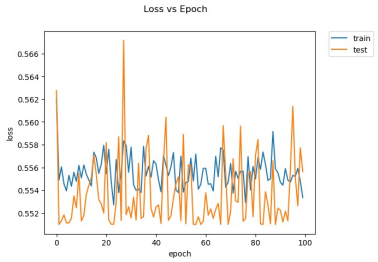


Figure 4: Our training and test loss curves before implementing batch normalization.

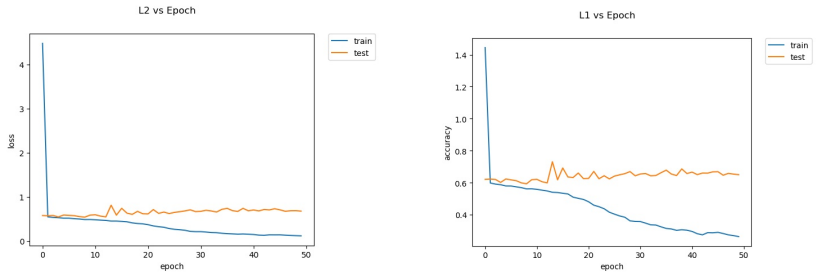
After implementing batch normalization, our training and test loss improved. Using a dropout of .2 helped to stabilize the learning curve in the training dataset, but the test dataset still didn't seem to be learning which can be visualized in Figure 5. This could be seen since the train loss plotted seemed constant. We also calculated out loss if guess naively by always guessing the average score of the trainset to have a benchmark to test if our model was learning anything nontrivial. This can be seen in Table 1. In comparison, without pretraining, the model wasn't learning anything, seen in Table 2.

Table 1: Naive Loss

Train	Test
0.604	0.627

Table 2: No Pretrain Loss

Train		Test	
L1	L2	L1	L2
0.263	0.115	0.676	0.650



(a) L2 loss

(b) L1 loss

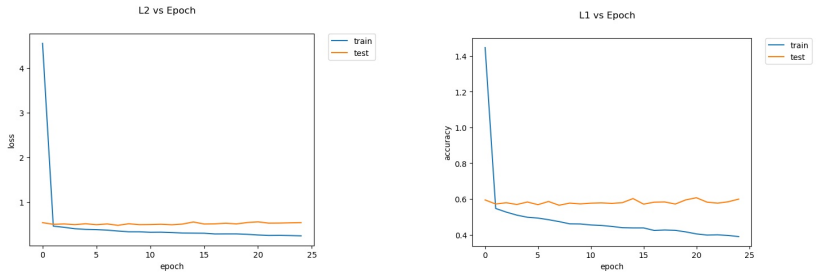
Figure 5: Our results after batch normalization and dropout, before pretraining.

After pretraining, the model can still be seen to overfit, but there is a bright side. At first glance, the results don't look much better as shown in Table 3 and Figure 6.

Table 3: Pretrain Loss

Train		Test	
L1	L2	L1	L2
0.390	0.242	0.599	0.538

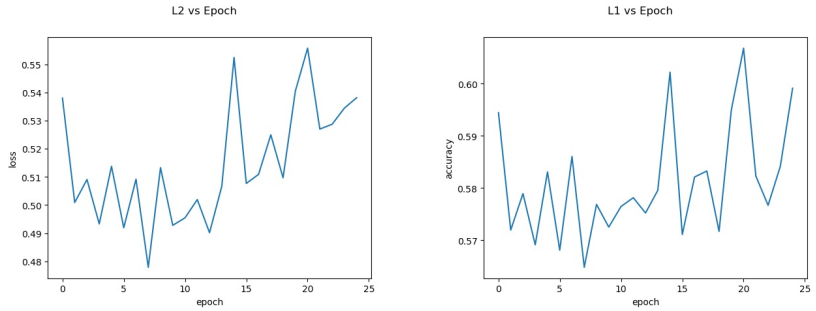
It's worth noting that the test L1 loss 0.599 is technically less than the naive test L1 loss 0.627 although barely.



(a) Loss (b) Accuracy

Figure 6: Our results after pretraining.

The interesting thing however can be seen when we run some more analysis on the test loss. If we just plot test loss without the scale of train loss, we can see the change over the epochs at a finer grain level in Figure 7.



(a) L2 Loss (b) L1 loss

Figure 7: Test loss plots after pretraining

The loss spiked down heavily before increasing slowly due to overfitting which may be due to overtraining. At it's lowest, the train loss was 0.565 at epoch 7. The corresponding train loss was 0.475. Both the train and test L1 losses are lower than the naive L1 loss which implies that at epoch 7, our model had learned something significant. These results are summarized in Table 8.

Table 4: Pretrain Loss at Minimum Test loss

Train		Test	
L1	Naive	L1	Naive
0.475	0.604	0.565	0.627

6 Discussion and Analysis

One big limitation of our model is how much it overfit to the training data. Since it is not able to generalize to any pictures of food and drinks, this means it is definitely not at a stage where it can generalize to image aesthetics overall.

6.1 Trainset

In the future we would have liked to explore how keeping the model architecture the same but using a different training dataset would have affected our results. Using pretrained weights gave us the best results, and the pretrained weights were trained on a very large dataset. This advocates for the claim that using a better dataset might lead to better results.

Better can be defined in many ways. First and foremost, it means bigger. Having more images would have helped our model generalize and would have decreased overfitting.

Better, in this case, also means more general. If we were able to get a wider range of images aesthetics, our data wouldn't be as narrow and the added generality may have helped the model. One issue is that these images were all submitted for a photography contest implying that they already represent a above average sample of photos in the world.

Additionally, using the larger dataset with all categories of images rather than just food and drink would, in theory, help to eliminate the limitation of our model to only predict scores for food and drink images. This points to the bigger issue that it is difficult for a model can't learn something a human can't predict themselves. In other words, if there is no true underlying distribution, the model will find it difficult to learn that. We explored this in section 2.2.2, but even using regression over classification isn't enough to overcome the very low variance of our data.

6.2 Dimensionality Reduction

To further address the persistent issue of overfitting, we could experiment more with different methods of dimensionality reduction. The relative success of the pretrained model also vouches for this idea. The pretrained model essentially gave us a form of dimensionality reduction. We would like to explore other menthods of dimenionality reduction and feature extraction to improve this model.

6.3 Related Goals

We will could also experiment with learning specific style attributes such as brightness, saturation, and definition and use those as feature encodings to see if it improves the accuracy over just using the RGB pixel array.

6.4 Future Applications

The dataset has 9 classes along with food and drink, each with their own train and test images. Thus this dataset alone can be used to train a model on many types of images, not just food and drinks. For example, people who want to upload images of their place to Airbnb could use this tool to estimate how well their photos will be received and people will want to book a stay. Or, this model could be used for people selling clothes on secondhand reselling apps to let the seller know whether they should consider uploading better pictures.

Another perhaps very interesting application is using this model as a preprocessing tool for data fed into other machine learning models. Finding good data is a pain point for any machine learning goal. This can help weed out poorly lit, unclear images from a dataset.

References

[1] Laufer, Jens. “Image Aesthetics Quantification with a Convolutional Neural Network (CNN).” Jens Laufer, 20 Mar. 2019, jenslaufer.com/machine/learning/image-aesthetics-quantification-with-a-convolutional-neural-network.html.

[2] Lu, Xin et al. “RAPID: Rating Pictorial Aesthetics using Deep Learning.” MM ’14 (2014).

[3] N. Murray, L. Marchesotti and F. Perronnin, "AVA: A large-scale database for aesthetic visual analysis," 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, 2012, pp. 2408-2415, doi: 10.1109/CVPR.2012.6247954.

[4] X. Fu, J. Yan and C. Fan, "Image Aesthetics Assessment Using Composite Features from off-the-Shelf Deep Models," 2018 25th IEEE International Conference on Image Processing (ICIP), Athens, 2018, pp. 3528-3532, doi: 10.1109/ICIP.2018.8451133.

dataset github repo: `https://github.com/mtobeiyf/ava_downloader`