The Classification of Food using Color Images

Author: Nicholas Evans

Email: nbe20420@uga.edu

Dataset: https://www.kaggle.com/kmader/food41

Abstract

The purpose of this project is to take different images of food items and attempt to classify them into categories. There are 50 different types of food used, with 1,000 of images per type, resulting in 50,000 images to sort through. I used the Convolutional Neural Network model to break a part the images and learn them piece by piece. During the preprocessing stage, each image needed to be resized and scaled. The labels for each image needed to be reformatted to create one hot representation. After experimenting with the layers, nodes, and attributes, the model was found to have a max approximate 25% accuracy reading for the images used.

1. Introduction

This project works to classify images based on the food items presented in each of the pictures. The goal for this project is to achieve the highest accuracy possible for classifying each type of food while maintaining a low value of error. The overall dataset has 101 different types of food; however, for this project, only 50 will be observed. Each type of food has 1,000 images, which gives a total of 50,000 images to sort through. Convolutional Neural Networks, or CNN, is the better model to use for classifying images because it is very easy to overtrain data and create a high accuracy and a high error for your training data. It is important to observe the validation set rather than the training data due the ability to overtrain. Steps are taken to make sure the model

does not waste time and run for longer than needed. Training the model, however, can only be half the work with a dataset this large. Splitting, scaling, and sorting the data can sometimes take just as long to train a model.

I begin by reading the images to obtain a list of all the different food types. To save on time, while reading the images, I also resize the images, so they are all the same. It is important to keep all the inputs as similar as possible to ensure all the inputs are constant. I shuffle the data and then separate the inputs from the labels. It is important to format the labels in one hot representation so that the model can evaluate the accuracy and loss. Finally, I scale the data to optimize the CNN model. Once I have all the data processed, it needs to be saved to not be repeated more than once.

After saving the data from processing the images, I load it back in to train the model. For a CNN, I begin with several convolutional layers starting with a smaller size layer and increase the size with each layer. The convolutional layer takes in the 3D shape that is an image and breaks it into pieces. After the convolutional layers, I need to flatten the 3D shape to use a fully-connected or dense layer. The dense layers help reshape the output of the layers in preparation for the final classification. All layers up to this point are using an activation function called rectified linear unit. This determines what is outputted after each layer. For the last layer, I use the function softmax because this turns each output from the layer into a probability for each class. This activation function is needed to calculate the loss and accuracy for the categorical nature of this model.

The basic format of a CNN is used, but there are many things you can change to attempt to make the model better. For example, you can change the amount of each type of layer and the size of the layer. You can also change a few things within each layer, such as how big of a window or

piece of the image each convolutional layer can look at one time. After changing almost everything from the start of training models, the results were not as promising as I thought they would be. The highest percentage of accuracy maintained after making changes is approximately 25%. This does not seem high, but it is much better than the very first model.

2. Data - Preprocessing

Originally, the dataset had a total of 101 different types of food; however, I only observed 50 images to have a base model to work with. The result was 50,000 images to preprocess and train. Due to the size, I had to take a few extra steps in preprocessing. Each image was dialed to 100x100 pixels. I decided to not greyscale the images, which made the images 3 times larger at 100x100x3. After reading them, it is important to shuffle the data, so the model learns the images and does not develop patterns in the input.

After shuffling the data, I can separate and scale the pixels. Generally, the normalize function would suffice; however, the size of the data would not allow to pass all the images at one time. Pixels can have values anywhere from 0 to 255. An easy way to scale these values is to simply divide them all by 255. This will place each value in between 0 and 1. To do this, I had to embed 4 loops to cycle through each pixel for each color image. This step gave the ability to scale the data without having to pass in all the images into a normalizing function.

The last bit of manipulation needed before putting together the model is formatting the labels. Generally, it is typical to just use indexes as the labels for each class. If you have more than 2 classes, then it is easier to calculate accuracy and loss through labels that are in one hot representation. Instead of having a 1-dimensional list that uses indexes that range from 0 to the

number of classes the model is trying to label, the one hot representation is a 2-dimensional array that uses only 1's and 0's. The shape of the array can be seen as: (# of inputs, # of classes). Each input has a list of 0's except for a single 1 that is designated for the class that it belongs to. For example, each input would be labeled as: [0,0,1, 0,…...0].

After preprocessing the data, it is highly important that it all be saved into a file, so this does not have to be repeated more than once. Dealing with large datasets, it can be very beneficial to save any preprocessed data so when experimenting with models you do not waste any time processing data repeatedly.

3. Experiments

After loading the data, I can build a CNN to train. I want to start off with some convolutional layers, and with this specific model, there are three. The first layer starts with 32 nodes, the second has 64 nodes, and the last layer has 128 nodes. Originally, I started with 16, 32, and 64 nodes, but after experimenting with the number, I decided to stay with 32, 64, and 128 due to the accuracy being higher. Each of these layers had a window size of (3,3), an activation function using rectified linear unit, and max pooling in that order. Each convolutional layer also has padding added to make sure the output matches the input so that it does not shrink. The max pooling has a window size of (2,2) which means out of the four different numbers in the window, only the highest number is retained for output. There is also a stride size of (2,2) to not have any windows overlap each other.

After I go through the convolutional layers, I flatten the output, so it is ready to go into fully-connected or dense layers. There are three dense layers. The first has 128 nodes, the second has

50, and the last one has 50. The last two have 50 nodes so it can prepare for the final output, which is classifying the images. The first two dense layers have rectified linear unit activation functions. They also have a dropout function. This is set to 0.2, so it drops out 20% of the data for both layers, which helps the model from overtraining. The last dense layer has a softmax activation function. A softmax activation function has an output of probabilities which is what is needed for calculating the accuracy and loss.

After I go through all the layers, I am ready to compile and fit the model. When compiling, I use the attributes, loss, optimizer, and metrics to specify how exactly the model should run and then calculate the results. Loss and metrics are not too interchangeable since I am dealing with multiple classes. Loss is set to "categorical_crossentropy" and metrics is set to “categorical_accuracy”. The optimizer deals with how the model learns, and for this project, I used “adam”. When I fit the model, there were 4 important attributes I needed to use: batch_size, epochs, validation_split, and callbacks.

I needed batch_size because like scaling our data, I needed to break up the images so I was not trying to pass in all of them at one time. I set batch_size to 32 so it would only call in 32 images at a time to run through the model. Validation_split splits the data into training and testing. This is important because if I use the entirety of the data, then the model will eventually memorize the images instead of learning how to classify new ones. Callbacks is used to retain the models’ data every time it is used and is used to affect the performance of the model. For instance, tensorboard is used with callbacks to measure how the model does over each iteration or epoch. Every time the model has run through all the images, it goes back and tweaks itself so it can perform better. This is called an epoch. Another callback I use is early stopping. This focuses on a particular metric of the model such as accuracy or loss. Whenever it stops going in the desired

direction, it stops the model early. For this project, I watch the validation loss and whenever it climbs instead of drops, the model stops after 5 more epochs. The final callback used on this project is a learning rate reducer. It also watches the validation loss and every time it rises instead of dropping, it reduces the learning rate to not overshoot the optimal accuracy. The final attribute to set is epochs. Since I have an early stop callback, I set the bar high at 50 epochs just in case the model does well. Otherwise, the model stops early when it starts to overfit the data.

4. Analysis

Every layer, attribute, and node used has been adjusted throughout the experiment to enhance the accuracy of the model. Other attempts at improving the model have not done so well and have not been added to the list of our final model due to the decrease in accuracy instead of increase that I worked for. Original attempts at the model had very low results at below 10% accuracy. However, the final model did better at approximately 25% accuracy. The model could possibly do better with more time committed to experimenting. Figure 1 is representation of how the final model did based on each epoch.

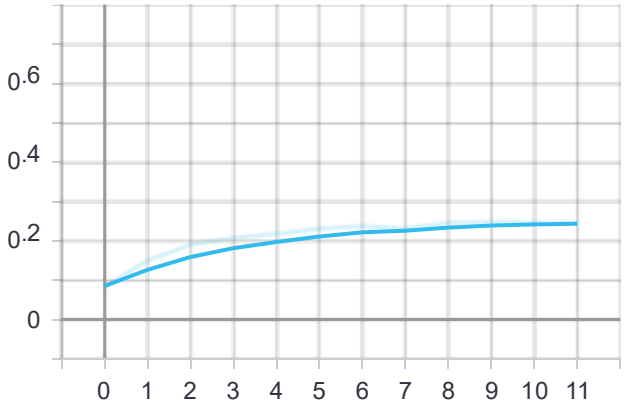

Figure 1: Representation of final model validation accuracy

You can see the accuracy rising over each epoch. The model stopped at 12 epochs because the early stop function read an increasing loss when it should be decreasing. This shows us that every image ran through the model 12 times has only a 25% chance of being classified correctly. In order to ensure that the data is not overtrained, I measure the loss of the validation set. As you can see in figure 2, at the 7th epoch, the loss begins to climb again ever so slightly. This is difficult to tell because of the reduction on the learning rate. When the loss rate increases, the learning rate decreases which causes the changes of loss to become smaller.

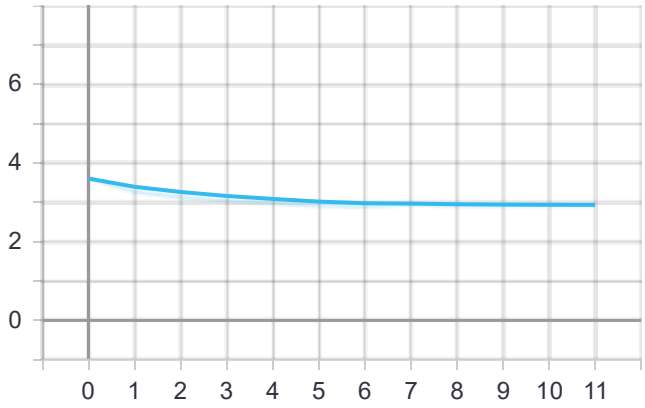

Figure 2: Representation of final model validation loss

Overall, the model did well to increase its accuracy overtime. It more than doubled while also minimizing the loss. After starting out with a low accuracy of 10% and ending with 25%, I believe the model did extremely well.

5. Conclusion

After starting out with a low accuracy of 10% and ending with 25%, I believe the model did extremely well. The loss could be reduced more, and the accuracy could increase more. The model predicts the right class of food only a quarter of the time, and I would like for that to increase to at least 50% or greater. This could probably be done by looking more into the

pictures themselves and trying to determine if there are any that can be disposed of or left out of the training model. There could also be more done with the layers of the model by changing the number or the size of the layers. I believe there is much room for this model to grow.