Food Image Classification and Model Analysis

Ty Farris, Sean Nesbit, Marine Cossoul
CSC 466 - Spring 2021

GitHub
Source Code

Background

Our team used the Food Images (Food-101) dataset to categorize different food images. The Food-101 dataset has 101 categories of foods from apple pie, to waffles with 1000 images in each category. This project will implement a classification model to predict the category based on a food image. This concept can be applied to a few different areas, such as recipe recommendation and nutrition identification.

Methods

Our first step was to break down the data into training and testing data to later be used with our Convolutional Neural Network. The initial attempt at this was by reading every single file path as the *X*, associating the directory label as the *y*, and then shuffling and splitting the data into training and testing subsets. However, since the CNN takes images as arrays of floats representing the RGB values and not as file paths, we realized we would have to use the keras ImageDataGenerator.

The Keras ImageDataGenerator allowed us to input our data directory path and would transform, scale, and apply other useful augmentations to the images as they were read by the neural network. This allowed us to scrap the previous file path based approach and let the ImageDataGenerator handle the bulk of the work.

We then built the architecture to our network based on a standard VGG-16 architecture. These CNNs are designed to produce good accuracies with limited modifications to the architecture and work well with lots of different image classification problems.

Problem 1:

The first problem we ran into was when passing the ImageDataGenerator to the CNN. When attempting to build and fit the network, our estimated per-epoch time was around 8 hours. At that time, we did not know that our Google Colab Notebook was not connected to a GPU and we were not getting those parallelized benefits. Our team attributed the slow performance to the 101,000 images being transformed by the ImageDataGenerator each time they were needed by the neural network. We realized that if we had to change our architecture or re-fit the model, we would spend far too much time waiting for the model to train which was not acceptable.

To solve this issue we decided to pre-process the data using OpenCV, save it to our Google Drive in a NumPy file, and read those files whenever we needed to reuse the data. Since our drive was not able to store the entire processed dataset, we saved only 200 files (instead of the available 1000) per class. This gave us a dataset of 101 classes with 200 files in each class.

Problem 2:

The next significant issue that arose was when we attempted to load the 101 classes with 200 files into memory. The free version of Google Colab does not have enough RAM on the CPU or GPU to hold our entire dataset of pre-processed images. While loading the data, once the RAM was exceeded, the notebook would crash and disconnect. This was extremely problematic as the way our data was stored, we could not batch our data into the model to solve this issue.

We decided we wanted to get some sort of tangible result, so we scaled down the amount of images to read out of each class from 200 images to 70. Combined with discovering how to attach our notebook to the Colab GPUs, we were finally able to load, train, and output a model.

Problem 3:

The third problem was that the model had a low accuracy around 6-8% because we decreased the number of images to 70 images per class. In addition to our low accuracy, our model was quickly overfitting. After our project status meeting, we realized that we should train on a subset of the dataset by reducing the number of classes. We decreased the number of classes from 101 classes to 20 classes and increased the number of images per class from 70 images to 200 images. This change increased our model accuracy to around 40%.

We originally were loading from a numpy file, but we could only store 200 images per class due to storage restrictions. We decided to switch to an ImageDataGenerator to load in more images per class. With now being attached to the GPU, we were able to run 20+ epochs in around 3 hours. The last iteration was to load in 20 classes and 1000 images per class which increased the model accuracy to 43%.

Model Architecture Revisions:

In order to increase the model accuracy and decrease overfitting, our team changed the following parameters: dropout, regularizers, convolutional layers, train/test/validation split.

1) Dropout
   We added two dropout layers near the dense layers in order to randomly ignore some neurons at a layer. The recommended range was 0.25 to 0.5 but there was no significant improvement once dropout was applied.

2) Regularizers

Regularizers are another way to reduce overfitting and we experimented with L1 and L2 regularizers. L1 regularizer estimates the median while L2 regularizer estimates the mean of the data. When we applied the regularizers to the dense layers, L2 had a slightly better performance than L1, but overall there was no significant improvement.

3) Network Layers
   We tried changing the pattern of the network by using a VGG network architecture, interchanging convolutional layers and maxpool layers, and changing the number of dense layers at the end of our model. We also changed the number of filters at each convolutional layer and the number of neurons at the dense layers. There were multiple model iterations, but the highest accuracy by changing the model parameters yielded around 41%.

4) Train/test/validation split
   We used to have a train/test split and then realized we needed a validation split for the model during training. We originally applied the validation_split parameter in model.fit() but saw no improvement, so we created our own train/test/validation datasets saved in numpy files. We then needed to add data augmentation so we made the train/test/validation split with an ImageDataGenerator.

Results

In order to test our model, we ran variations of the image dataset and compared the training and validation accuracies of each variation. We leveraged keras ImageDataGenerator to implement the following image modifications: gray-scaling, rotation, brightness, and image size.

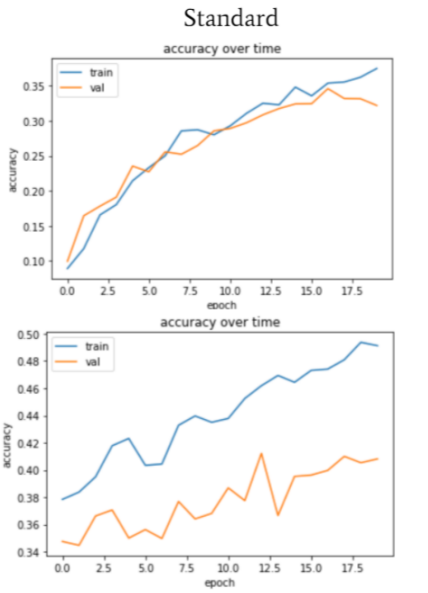
The following is the model architecture that produced the best accuracy of 43% and is used across the following experiments.

```
import tensorflow as tf
tf.keras.backend.clear_session()
num_classes = 20

cnn = tf.keras.models.Sequential()
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(5,5), activation='relu', input_shape=[img_height, img_width, 3]))
cnn.add(tf.keras.layers.Conv2D(filters=32, kernel_size=(5,5), activation='relu'))
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
cnn.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
cnn.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), activation='relu'))
cnn.add(tf.keras.layers.MaxPool2D(pool_size=2, strides=2))
cnn.add(tf.keras.layers.Flatten())
cnn.add(tf.keras.layers.Dense(128, activation='relu'))
cnn.add(tf.keras.layers.Dense(num_classes, activation='softmax'))

cnn.summary()
cnn.compile(optimizer='adam', loss="categorical_crossentropy", metrics=["accuracy"])
```

Baseline Run

We first ran our model on the standard set of images as a baseline for training and validation accuracy. The following transformations were consistent throughout all the tests: 128x128 size, sheer, zoom, and flip. The results of this run are illustrated in the figure to the right.

As we can see, the model begins to overfit around the 17th-20th epoch at around 32% validation accuracy, but the validation accuracy continues to climb and then begins to flatten at around 40% accuracy. This is the baseline with which we will measure the usefulness of different image modifications in model accuracy.

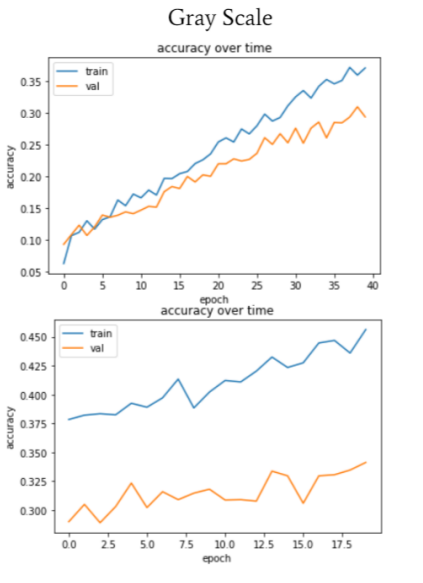


40 Epochs: 49% Train | 40% Validation

Grey-Scaling

We then implemented a grey-scaling modification on the image dataset to observe its effect on model accuracy. The following figure illustrates the accuracy over time.

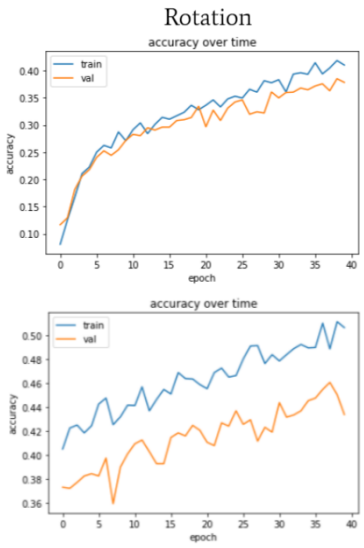
Here, we can see that the training and validation accuracies follow a similar trend to the baseline, beginning to overfit a bit sooner at around 25% validation accuracy. Grayscaling decreases the learning rate significantly, but key features are lost which reduce overall training and validation accuracy. Compared to our baseline model, this decrease in accuracy would be caused by the fact that color plays a significant role in classifying foods.



60 Epochs: 46% Train | 34% Validation

Rotation

Next, we modified the images by rotating by 30° repeatedly over 360°. This meant 12 rotations per image, causing longer training over more epochs as can be seen in the following figure.

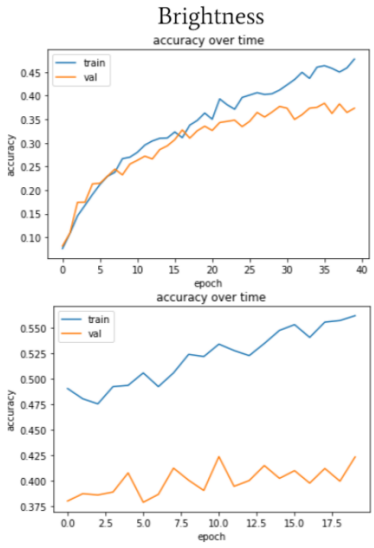


Although the rotation run took significantly longer than other tests (80 epochs), it yielded the highest validation accuracy and began to overfit after 40 epochs, much later than the other modifications. This high accuracy is because rotating the image could allow the model to capture different features of the data as a result of varying angles.

80 Epochs: 50% Train | 43% Validation

Brightness

We then tested the model's accuracy levels on images with varied brightness. We chose a scale of 40-150% brightness and found the following results:



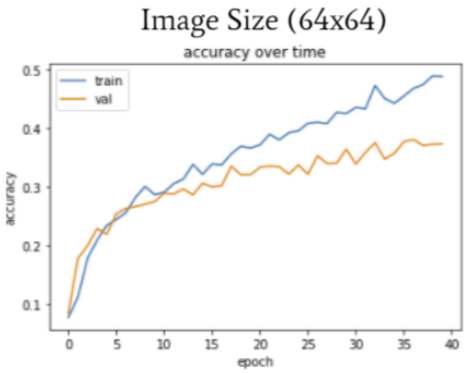
This image modification produced higher overall training and validation accuracies; we presume this is because the normalization of brightness values reduces image categorization based on environmental lighting and instead prioritizes the characteristics of the food itself.

60 Epochs: 56% Train | 42% Validation

Image Size

Finally, we modified the image sizes from 128x128 to 64x64 to observe the effects of image size on model accuracy.

Decreasing image size from 128x128 to 64x64 has a negative effect on validation accuracy, but reduces number of epochs and training time per epoch as the data size is significantly smaller. The results show that the model does not capture more features and data from a compressed version of the image.



60 Epochs: 42% Train | 34% Validation

Discussion

After looking at related work for different model architectures, existing VGG architectures rarely got above an accuracy of 70%. Elite Deep Learning and SENet reached above an accuracy of 80% [1]. The architectures that achieved around 70-80% accuracy had a complex architecture and used larger datasets, such as 756 classes to train on. With our limited time and resources, we were not able to mimic this architecture.

Overall, our model currently overfits and has a lower accuracy of around 43%. This accuracy makes sense as existing related works had around 70%-80% accuracy with the Food101 Kaggle dataset. Shrinking the number of classes and adding more data per class had the biggest effect on our model accuracy which increased from 8% to 41%.

After the Project Status Meeting, our direction changed from attempting to maximize our overall accuracy, to understanding which ImageDataGenerator augmentations cause an increase or decrease in model accuracy. We discovered simple parameters had significant effects on the accuracy of models with identical architectures and estimated the causes of these discrepancies.

Future steps would be to get more storage in order to load more of the dataset and reduce our underlying issue of overfitting. Another option would be to use a different dataset that has been shown to produce a higher accuracy and see whether the change in ImageDataGenerator parameters affect the dataset the same. A third option would be to examine how different model architectures affect accuracy given the various ImageDataGenerator parameters.

Citations

[1] Gjertsen, K. (2020). *Food Classification with CNNs* (thesis).