Recognizing Korean Food in Photos through Transfer Learning of Deep Features

Joseph Kim

March 9, 2016

Introduction

Food is one of the most common things that people take photos of these days. Google recently reported that food photos were the second most common kinds of photos uploaded by Google Photos users [1]. Given that there is so much food photos data, as well as the fact that there are so many food-based startups these days, applying image recognition on photos of food has lots of potential for interesting applications, such as recommender systems. A quick google search shows that there are a few academic food recognition projects and papers, but none specifically for Korean foods.

The primary goal of this project was to create a Korean food recognizer that can identify Korean foods in photos. Specifically, the goal was to create a supervised machine learning model trained on images of Korean foods that takes a new image as input, and accurately output the name of the dish in the photo. Because of the success of deep convolutional neural networks (CNN) for image recognition in recent years, a good candidate model would be a CNN that is trained on Korean food photos.

Because training all the weights in every layer of a CNN from scratch requires a lot of images and a lot of computational power, transfer learning can be used to reduce both the CPU workload as well as the size of the dataset. By applying transfer learning on a CNN already trained to recognized images, all that is left is to retrain the top layer of a pre-trained CNN to recognize Korean food in photos. Google's Inception-v3 CNN was a very easy choice for a pre-trained network, which had been trained on ILSVRC2014 dataset to perform very well on the ImageNet Large Visual Recognition Challenge. At the time of this writing, it outperforms all the other models for the ImageNet dataset [2]. The ILSVRC dataset contains 1.2 million images from 1000 categories [3], which means that Inception v3 was a good choice for a generic image recognition tool that I can fine-tune for this problem.

The deep features generated by a trained, state-of-the-art CNN for recognizing images in the second-last layer would be a good "embedding" of each image, which is why transfer learning works. Therefore, those features can be used to train other machine learning algorithms, such as support vector classifiers and logistic regression models. In order to set a benchmark performance, these other machine learning algorithms were trained on these deep features. In addition, another CNN implemented by a private company, MetaMind, was used for comparison.

The secondary goal of this project was to create a proof of concept prototype that can demonstrate a typical use case for this technology. Many machine learning projects come to completion where the end results consist of metrics and charts, with no real products. This is understandable due to the nature of how the projects are evaluated. However, with this project, there was an opportunity to develop software that can actually utilize the end results of the machine learning process for a tool closer to an actual product that people can use. Because "Foodies" these days take photos using their mobile devices, I wanted to show how food identification can work on a mobile device. This resulted in two other GitHub projects, kfood-server [4] and kfood-android [5].

Metrics

For this supervised multi-class classification problem, the performance of the models were primarily measured by *accuracy* on the test data, as defined by the following formula:

$$accuracy = \frac{TP + TN}{TP + FP + FN + TN}$$

TP is true positives, TN is true negatives, FP is false positives, and TN is true negatives.

For training the last layer of the deep convolutional network, the objective loss function is *cross entropy* (also known as *log loss*) defined as:

$$logloss = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K} y_{ij}\log(p_{ij})$$

N is the number of training samples, K is the number of classes, which is the number of Korean food dishes in this project, y_{ij} is the correct binary value (1 if it is that label, 0 otherwise) for the *jth* label for sample i, and p_{ij} is the predicted probability of sample i for the *jth* label.

Fine-tuning support vector classifier and logistic regression models using cross-validated grid-search also used *cross entropy.*

F1 score was also considered for analysis, but because the distribution of training data across each of the labels was not overly skewed (See Figure 1), *accuracy* was good enough to gauge performance for this multi-classification problem.

Analysis

Data

Acquiring and Filtering

20 Korean dishes were selected based on how common and popular they are among Korean and Korean Americans. For each of the dishes, 150-300 images were downloaded from the web using Google image search, for a total of around 5,000 images. Then each of the images were manually viewed, and filtered out if the images were low quality or deemed non-representative of the actual dish. After the filtering process, the dataset consisted of 3,470 images, with max, mean and min of 219, 173.5 and 112 images per label, respectively.

Table 1: 20 Korean Food Categories Counts

Name	Number of samples
galbijjim	219
kimbab	217
bibimbab	214
hotteok	210
nangmyun	207
dakgalbi	205
sullungtang	193
japchae	193
bulgogi	183
samgyupsal	173
bossam	173

Name	Number of samples
dakbokeumtang	171
jeyookbokkeum	165
samgyetang	158
ddukbokee	150
lagalbi	148
jeon	134
kimchi	127
ramen	118
yookgyejang	112

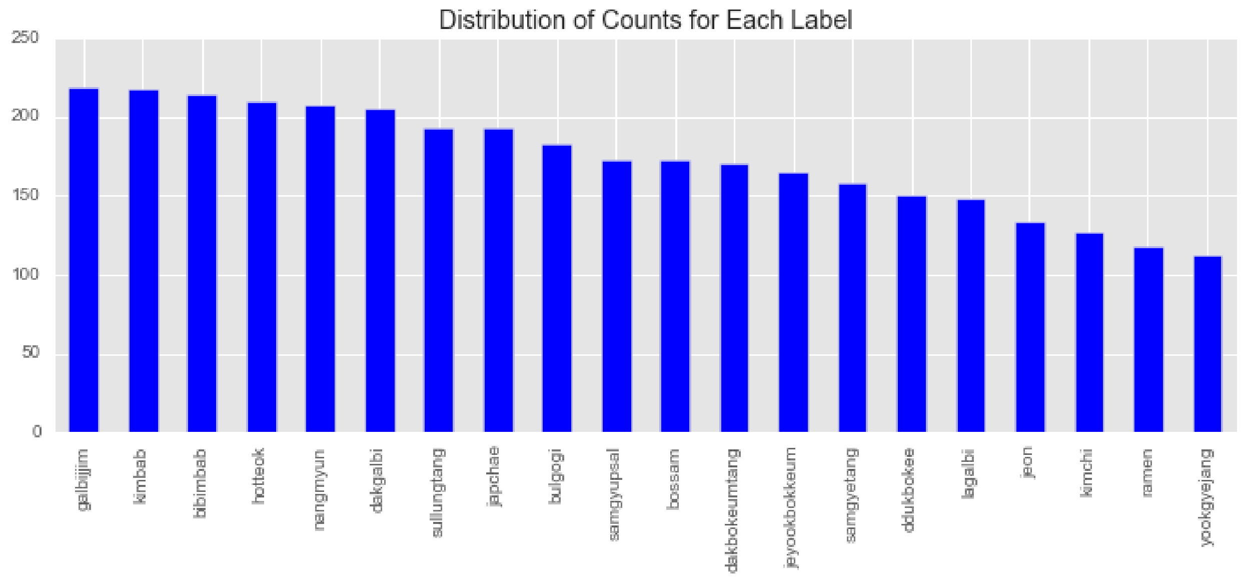

Figure 1: Data Distribution

Deep Features Transformation

Each image file was transformed to deep features, which are numeric embedding vectors of size 2048 using a modified version of the `classify_image.py` script provided by Google in the TensorFlow source code and saved to disk for further analysis. The embedding is the value of the tensor in the second-last layer of the Inception-v3 Deep Convolutional Network. Table 2 shows that the values of the deep features were similar to each other in terms of range, and there were no concerns about certain features dominating other features with values that were much bigger. In other words, standardizing columns was not a terribly big concern.

Table 2: Description of First Five Deep Features

	feature0	feature1	feature2	feature3	feature4
count	3470	3470	3470	3470	3470
mean	0.292122	0.409072	0.527338	0.205257	0.341956
std	0.210247	0.310620	0.348090	0.196429	0.309863
min	0.002776	0.000000	0.009342	0.000000	0.000000
25%	0.135991	0.177316	0.271595	0.066020	0.121058
50%	0.244641	0.341317	0.450077	0.147333	0.259115

	feature0	feature1	feature2	feature3	feature4
75%	0.396790	0.555601	0.708050	0.281080	0.469292
max	1.751740	2.421970	2.682040	1.811220	3.519750

Train-Test Split

The dataset was then split into separate train and test sets with the training set consisting of 70% (2431 samples) of the data, and the test set 30% (1039 samples). The splits were stratified with respect to the labels, so that the representation of the labels after the split remain similar to the dataset before the split.

Table 3: Stratified Train-Test Split

	train	test	% test
bibimbab	150	64	29.9
bossam	121	52	30.0
bulgogi	128	55	30.0
dakbokeumtang	120	51	29.8
dakgalbi	144	61	29.7
ddukbokee	105	45	30
galbijjim	153	66	30.1
hotteok	147	63	30
japchae	135	58	30.0
jeon	94	40	29.8
jeyookbokkeum	116	49	29.6
kimbab	152	65	29.9
kimchi	89	38	29.9
lagalbi	104	44	29.7
nangmyun	145	62	29.9
ramen	83	35	29.6
samgyetang	111	47	29.7
samgyupsal	121	52	30.0
sullungtang	135	58	30.0
yookgyejang	78	34	30.3

Algorithms

Transfer Learning

The Inception-v3 network was used to generate 2048 deep features, and the top fully-connected layer of the network was retrained, which applies the Softmax function to output probability values for each of the Korean food labels.

Transfer learning in machine learning is the application of knowledge learned in one problem to another similar or related problem, and has been around for at least a couple of decades [6]. A relatively recent discovery of transfer learning through trained deep neural networks has raised a lot of excitement and opened doors to a lot of applications. Researchers found that the trained weights of nodes in a trained network, particularly the lower and mid-level nodes, can be copied readily to other deep networks, along with the structure of the deep network, and just retraining the high-level classification layer can result in a network that performs very well for a new problem. For example, for image recognition, taking a convolutional neural network that performs very well on recognizing various types of images and retraining the classification layer to identify a specific set of labels such as dog breeds or food types can work well. That's because the lower nodes and layers of the CNN learn low-level features of the images, which are necessary for many image recognition tasks, but the higher nodes and layers learn high-level features, which tend to be more specific to the dataset [7].

A big advantage of transfer learning is that we can drastically reduce the total amount of computations needed to train deep networks. Successful deep learning projects are often trained on high performance clusters and servers with GPUS to reduce the time to compute all the vectorized matrix operations, and runtimes can be hours, days and even weeks. With only a personal laptop, an engineer can take one of these trained networks, and just retrain the last layer of the network on a smaller set of data for a specific classification task without performance or memory issues.

A disadvantage of transfer learning is that the resulting classifier model may not be able to outperform a CNN whose entire set of weights are retrained at all levels for the specific problem, not just the upper layers. A good way to leverage a trained CNN would be to take the weights and use them as initial values, but keep them variable so that the model can be fine-tuned for the specific problem. As long as there is enough good-quality training data that won't cause the new CNN to overfit to the training set, retraining the entire network may be a better idea if the computing resources are available.

Benchmark

Non-Neural Network Algorithms

Preliminary benchmarks were set using six different kinds of machine learning classifier algorithms, and 3 highest performing algorithms were fine-tuned to set a benchmark to beat.

Table 4: Stratified Classifier Types

Classifer Types
Support Vector Classifier
K-Neighbors Classifier
Random Forest Classifier
Logistic Regression
Gradient Boosting Classifier
AdaBoost Classifier

The initial performances of the non-neural network algorithms are described in Table 5.

Table 5: Preliminary Benchmark of Non-Neural Network Algorithms

Algorithm	Train Accuracy	Test Accuracy
SVC linear kernel, one-vs-one	1.000000	**0.794995**
SVC polynomial deg 2 kernel, one-vs-one	0.266557	0.271415
SVC polynomial deg 3 kernel, one-vs-one	0.063348	0.064485
SVC rbf kernel, one-vs-one	0.763472	0.704524
SVC linear kernel, one-vs-rest	1.000000	0.769971
SVC linear poly deg 2 kernel, one-vs-rest	0.839572	0.721848
SVC linear poly deg 3 kernel, one-vs-rest	0.868367	0.726660
SVC rbf kernel, one-vs-rest	0.830111	0.730510
KNeighbors, 3 neighbors	0.815714	0.619827
RandomForest, 10 estimators	0.996709	0.457170
LogisticRegression ovr	1.000000	**0.793070**
LogisticRegression multinomial	1.000000	**0.803657**

The 3 highest preforming algorithms from Table 5 that were selected for fine-tuning were SVC (Support Vector Classifier) with a linear kernel, Logistic Regression using multiple one-vs-rest classifiers, and Logistic Regression using a single multinomial model.

After fine-tuning the parameters of the three models using cross-validated grid search, the performance of the models on the test set improved marginally.

Table 6: Fine-Tuned Model Accuracy On Test Data

Algorithm	Preliminary	After Fine-tuning
SVC	0.794	0.797
LogisticRegression ovr	0.793	0.797
LogisticRegression multinomial	0.804	0.806

MetaMind

MetaMind, a private startup specializing in deep learning services, provides limited free services image classification. Their API allows users to upload test datasets to train a small dataset for transfer learning on a convolutional neural network, and then submit data to get predictions. With the train-test split data, I trained a classifier and ran test predictions to measure the performance of the service. The training accuracy was 79%, and test accuracy was 0.77.

Methods

Implementation

The data analysis and machine learning algorithms were run on a Jupyter Notebook running Python 2.7. The non-neural network algorithms were run using the Scikit-learn library, and TensorFlow was used for running deep learning algorithms.

Running TensorFlow's `classify_image.py`, based on the TensorFlow documentation [8] did not work for me out of the box. My initial TensorFlow installation was version 6.0, but the tutorial made use of a library that was not implemented at the time. However, upgrading to a newer version 7.0 caused another error. I was able to find an easy fix by staying with version 6.0, and modifying `classify_image.py`. The details are outlined in this StackOverflow post, where my answer is marked as the solution.

During the course of the project, I learned that the TensorFlow team provided a tutorial on retraining the Inception-v3 network's final layer for custom categories and images [9], so I was able to use that tutorial and the tutorial's python script `retrain.py` provided in the TensorFlow source code for this project.

TensorFlow's default installation guides do not provide a way to install TensorFlow on OSX with CUDA support. Because my laptop has a discrete Nvidia graphics card, I wanted to see how much performance improvements I would get if I compiled TensorFlow on my laptop with CUDA support. After compiling TensorFlow on my laptop using a step-by-step guideline I found online [10] and modifying `retrain.py` to use GPU for some of the matrix operations, I measured the time it took to train using the hyperparameters. Unfortunately, I did not get any speedups. In fact, the version compiled with CUDA ran slightly longer. In the end, I reverted to the default version 6.0 installation.

Table 7: Performance Comparison of TensorFlow w GPU

Training steps	Default Install	Compiled w CUDA
2000	2m18.055s	2m30.295s
1000	1m11.341s	1m19.523s

For the mobile application, I tried following the tutorial provided by TensorFlow[11], but I wasn't able to get my Android device to run the TensorFlow Demo app. I think this was due to the fact that my device is fairly old (bought over 3 years ago). Therefore, the next best thing was to create a RESTFul web server running TensorFlow so that mobile devices can send photos to the server, and the server can send back the classification label. This way is actually more scalable and better since most devices, even really old ones, have cameras and networking capabilities. Not only that, iOS devices can take advantage of a service like this as well.

The web server was built using Django running on Python 3.4. Additional Python packages include TensorFlow 6.0 and the Django REST Framework, whose usage I had to modify to provide API endpoints serving non-database model objects. The server is actually up an running on Heroku's free sandbox Dyno. A simple HTML interface is available at https://kfood.herokuapp.com/photos and a web service API is accessible at https://kfood.herokuapp.com/api/classify.

The android app was developed using Android Studio by IntelliJ using Android SDK 24.4.1 for devices running at least API level 15.

Refinement

Refining the TensorFlow model is a process of searching for the set of hyperparameters that would result in the highest test accuracy. The `retrain.py` script provides ways to do this with hyperparameters that can be inputted as arguments to the script. The following hyperparameters are available: training steps, learning rate, train batch size, validation percentage, testing percentage

The script also provides ways to distort the input image files in order to increase the effective size of the dataset, such as random horizontal flipping of the images, random cropping, random scaling and randomly changing the brightness of the images. However, these require image manipulations that are very intense, and when I tried enabling these parameters, I was never able to finish processing a single run of the `retrain.py` script. The process ran for several hours, and my laptop began to heat up, so I stopped it.

Because `retrain.py` randomly selects validation and test subsets for each run, I fixed the validation percentage to 5% and testing percentage to 25%. This was in order to keep it similar to the way the non-neural network algorithms were evaluated, where the test data consisted of 30% of the entire data set. Then, I tried various combinations of the number of training steps, learning rate, and train batch size to find the set of hyperparameters that would result in the highest test accuracy.

Table 8 shows some examples of the runs to search for the optimal set of hyperparameters.

Table 8: Hyperparemeter Search

train steps	learn rate	train batch	train acc %	valid acc %	test acc %	runtime
5000	0.01	200	92.5	74.0	77.4	10m26.148s
5000	0.01	175	92.6	74.0	80.2	8m51.950s
5500	0.01	175	95.4	74.0	76.0	10m28.258s
5800	0.01	175	90.9	68.0	80.2	11m7.562s
5900	0.01	175	94.9	76.0	79.0	10m36.280s
6000	0.01	175	94.3	69.0	81.4	11m11.572sp
6100	0.01	175	93.7	73.0	77.4	12m26.308s
6500	0.01	175	95.4	75.0	75.6	12m32.043s
7000	0.01	175	96.6	76.0	79.2	13m23.146s
5000	0.01	150	95.3	80.0	77.8	7m46.747s
5000	0.01	125	98.4	77.0	78.2	6m35.836s
5000	0.01	100	96.0	67.0	81.2	5m25.950s
5400	0.01	100	92.0	73.0	78.4	6m27.017s
5450	0.01	100	91.0	67.0	79.0	6m27.963s
5500	0.01	100	93.0	78.0	82.0	6m45.637s

train steps	learn rate	train batch	train acc %	valid acc %	test acc %	runtime
5500	0.01	120	96.7	78.0	**82.4**	7m35.594s
5550	0.01	100	95.0	80.0	80.6	6m37.738s
5600	0.01	100	95.0	73.0	80.2	6m52.054s
6000	0.01	100	96.0	78.0	78.8	6m33.281s
7000	0.01	100	96.0	72.0	79.6	7m39.502s
8000	0.01	100	97.0	78.0	79.8	8m39.124s
10000	0.01	100	97.0	71.0	78.6	10m49.417s

Results

The final retrained TensorFlow model had a test accuracy of 82.4%, which is a little bit better than the benchmarks, whose accuracies were around 80%. It performed better than the MetaMind model, which had accuracy of 77%. In general, most of the TensorFlow runs resulted in very similar ranges of around 80%, and small perturbations in the hyperparameters doesn't affect the model performance greatly. The best hyperparameters are shown in table 9.

Table 9: Optimal Hyperparemeters

hyperparameter	value
training_steps	5500
learning_rate	0.01
train_batch_size	120

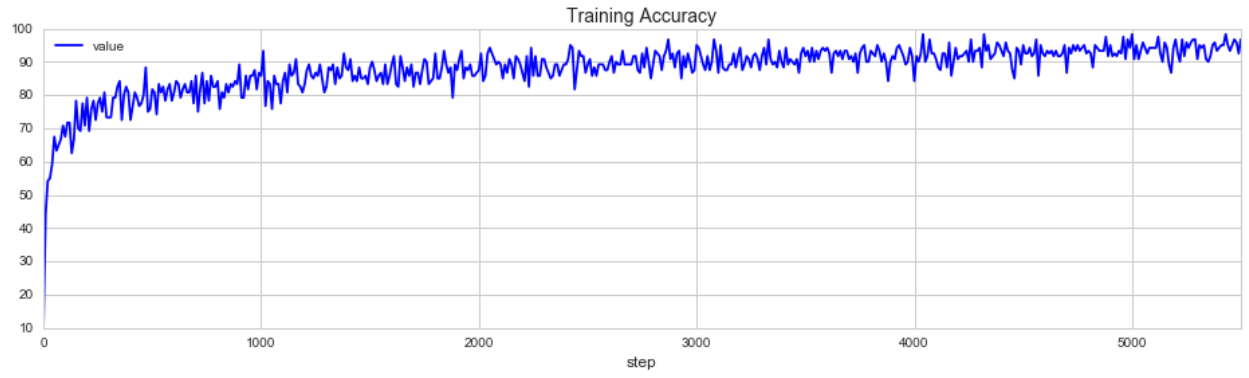

Figure 2: Training Accuracy

The resulting tensorflow model was saved to disk in a protocol buffer format, which is loaded by the web server.

The android device takes photos, sends it to the web server, and receive the classification label for the food image. Figure 5 shows the screenshot when the device receives back the classification label from the server.

Discussion

The results of 82.4% accuracy on the test set is pretty good given that there wasn't a lot of training data and that only a single layer was trained. I think that the performance can be improved greatly if instead of

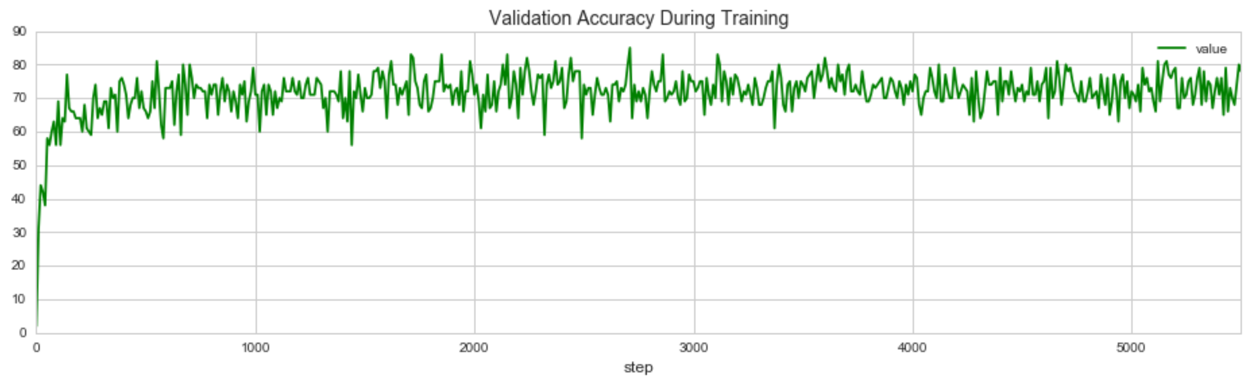

Figure 3: Validation Accuracy

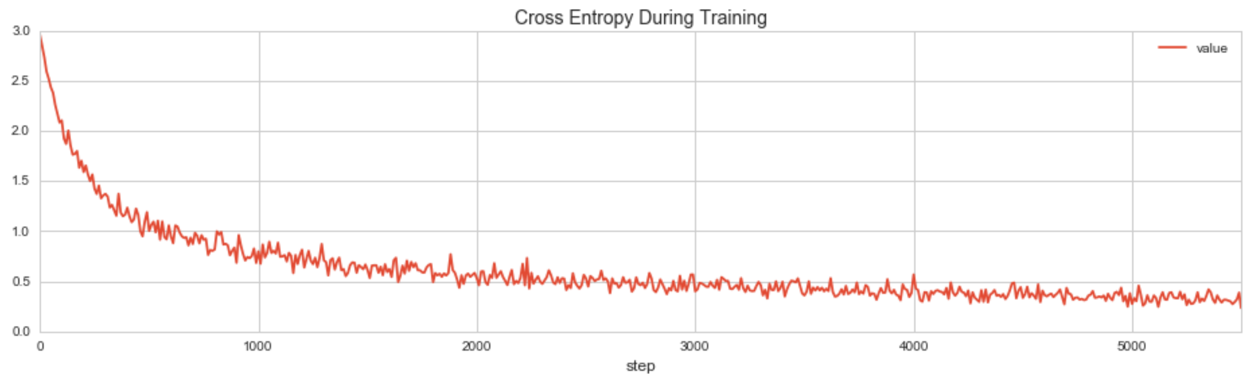

Figure 4: Cross Entropy

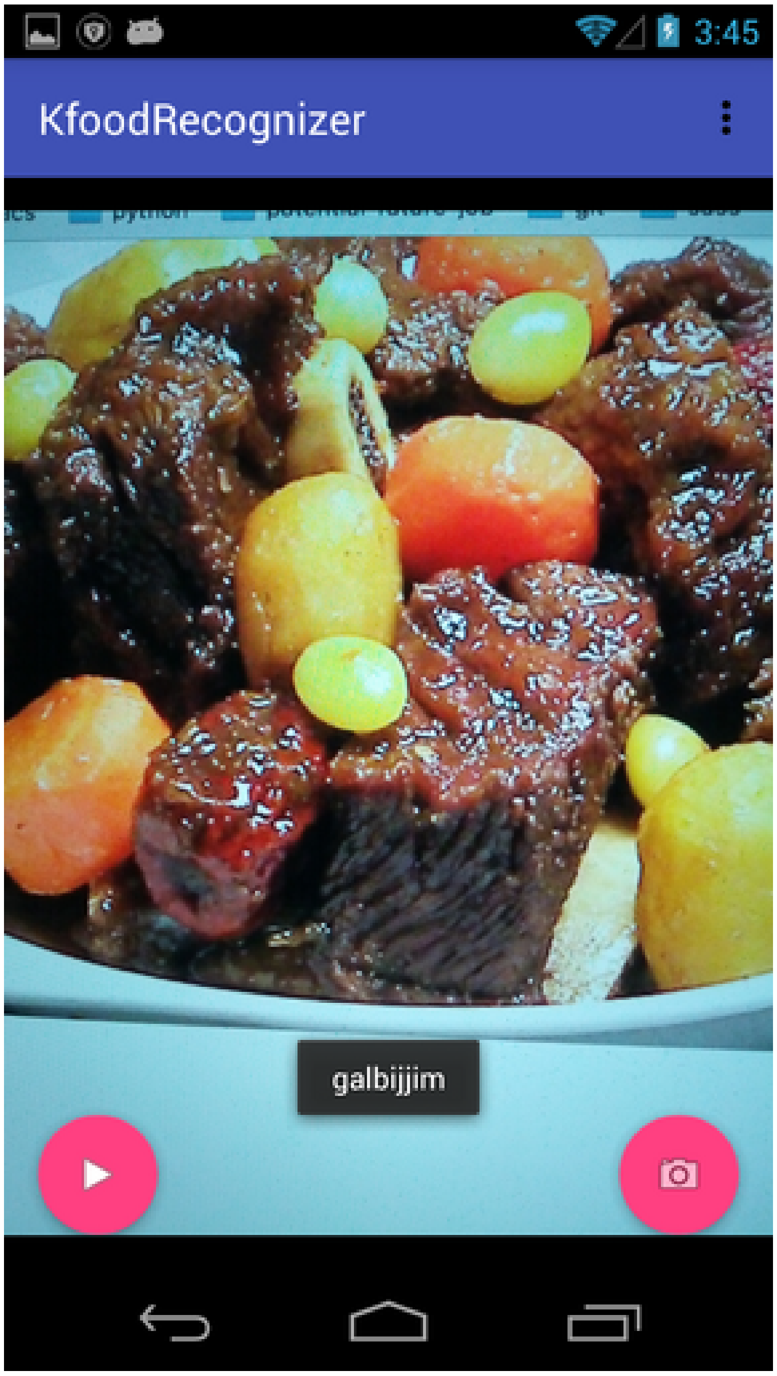

Figure 5: Android App Screenshot

just retraining the last layer, the entire network is retrained, with the Inception-v3 weights as initial values. Also, having more data would probably increase the performance by a few percentage points.

Figure 6 shows a confusion matrix of the predictions run on the test set. Here we can see that similar foods are mis-labeled. For example, sullungtang and samgyetang are two kinds of soups that are white-ish in color, and the retrained model tends to misclassify one for the other. This kind of mistake can actually be made by humans also.

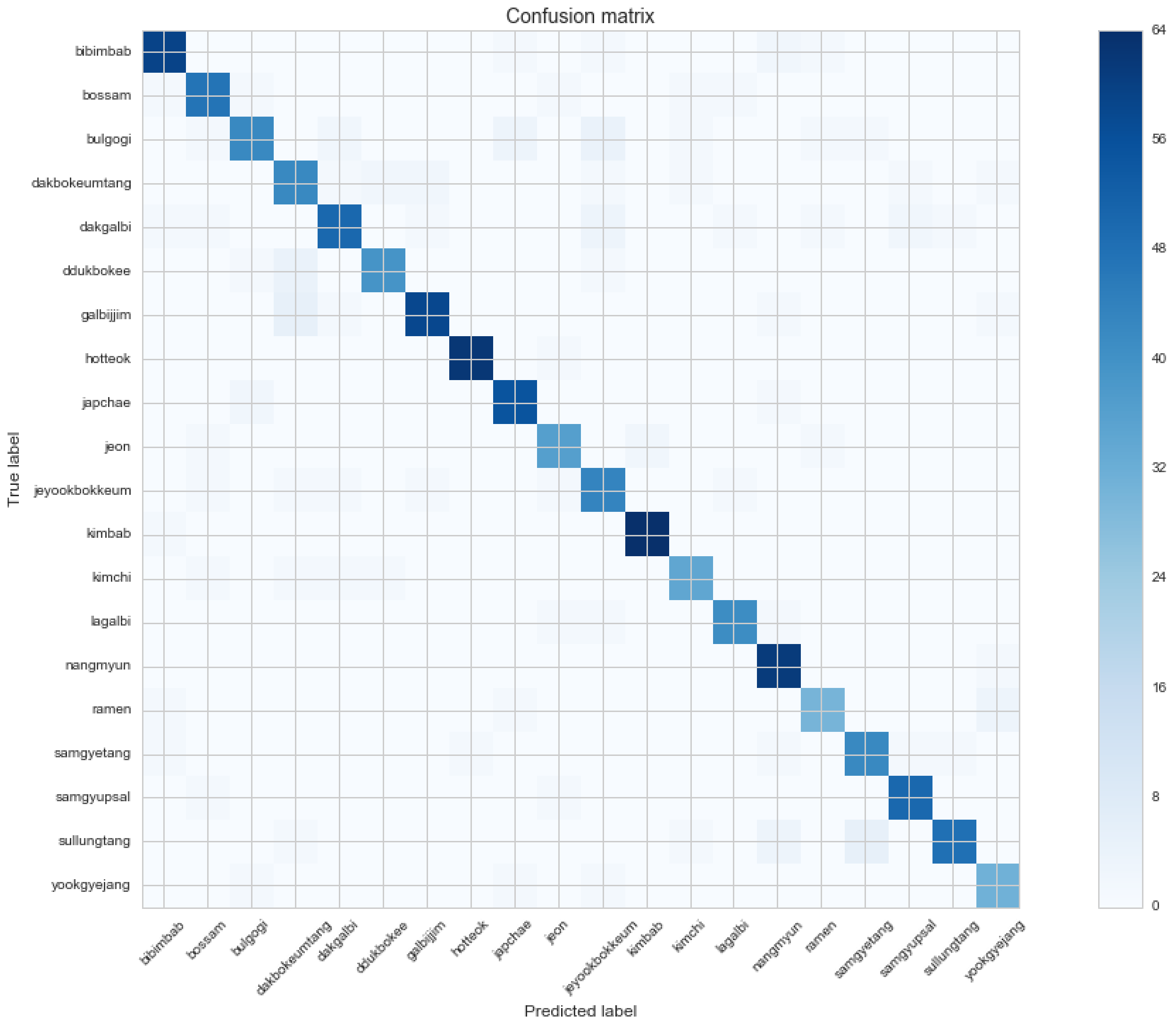

Figure 6: Confusion Matrix on Test Set

Image recognition can and is already being used for a variety of real-world applications. Transfer learning deep convolutional networks, opens doors to even more applications that can be developed quickly. Possible applications include food recommender systems based on images for food apps where searches on images can be run on a database of images or image embeddings to search for restaurants that have the food that the user wants.

Figure 9 shows that the image embeddings are spatially clustered in coordinate space, meaning that the deep feature embeddings can be used for identifying similar images and filtering, such as an app that can help find lost dogs where users can upload photos of stray dogs and it can be matched with a database of lost dogs. The spectrum of areas that can apply this technology is very wide, but the core technology is the

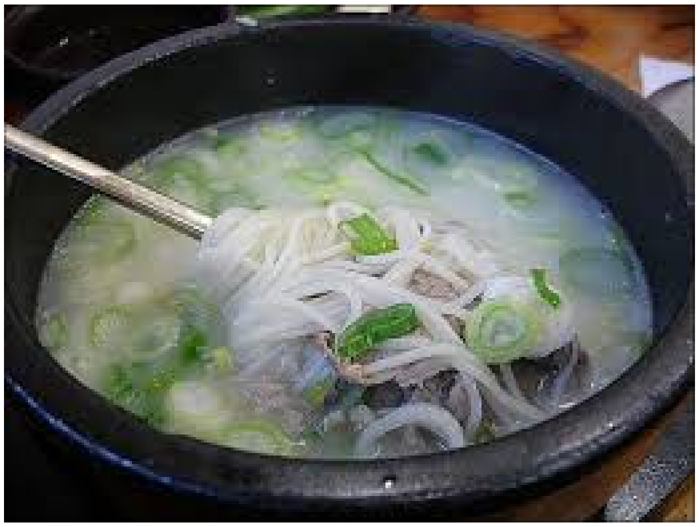

Figure 7: Sullungtang (Beef Broth Soup)

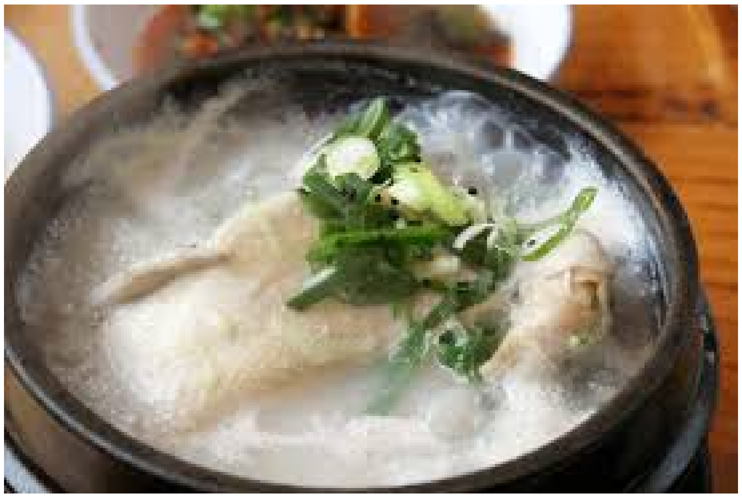

Figure 8: Samgyetang (Ginger Chicken Soup)

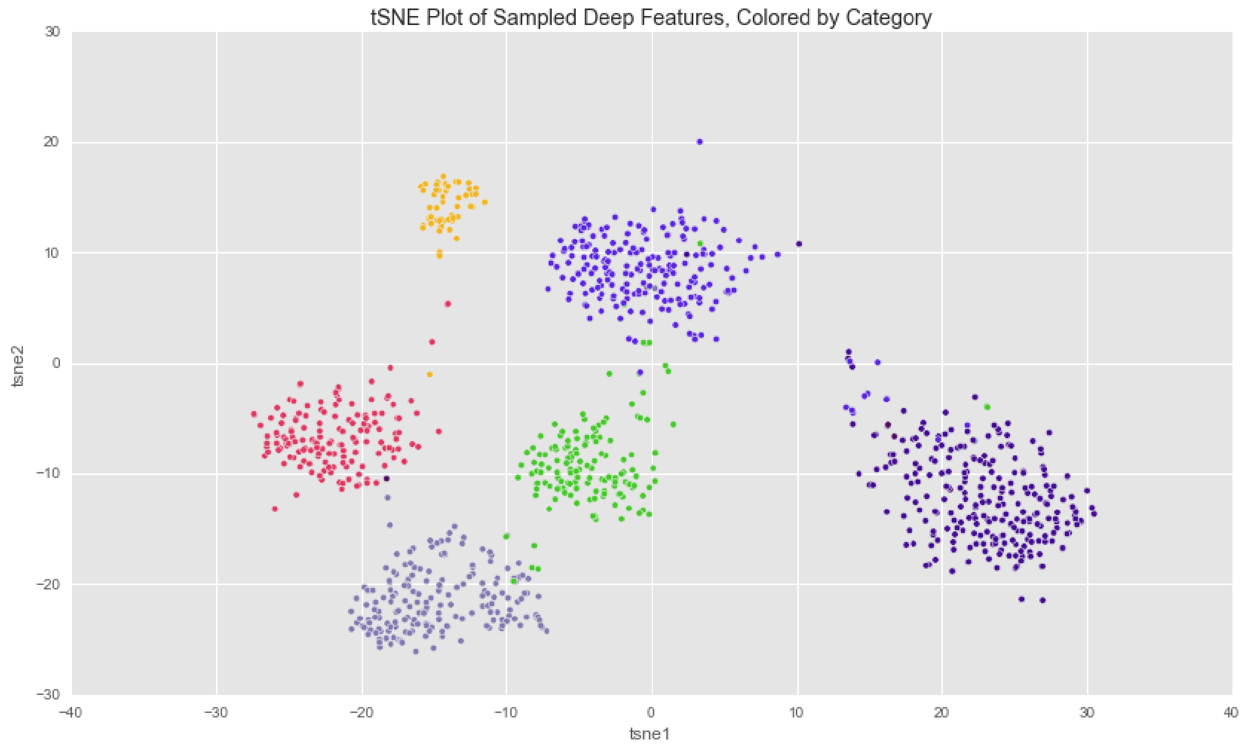

Figure 9: 2D t-SNE Plot of Deep Features

same. That's very exciting!

References

[1] C. Perry, "11 things to know about google photos," *Google Official Blog*. 2015 [Online]. Available: https://googleblog.blogspot.kr/2015/10/11-things-to-know-about-google-photos.html

[2] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015 [Online]. Available: http://arxiv.org/abs/1512.00567

[3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[4] J. Kim, "Kfood-server," *GitHub repository*. https://github.com/jjinking/kfood-server; GitHub, 2016.

[5] J. Kim, "Kfood-android," *GitHub repository*. https://github.com/jjinking/kfood-android; GitHub, 2016.

[6] Wikipedia, "Inductive transfer — Wikipedia, the free encyclopedia." 2016 [Online]. Available: https://en.wikipedia.org/wiki/Inductive_transfer

[7] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" *CoRR*, vol. abs/1411.1792, 2014 [Online]. Available: http://arxiv.org/abs/1411.1792

[8] "Image recognition," *TensorFlow Documentation*. [Online]. Available: https://www.tensorflow.org/versions/master/tutorials/image_recognition/index.html

[9] "How to retrain inception's final layer for new categories," *TensorFlow Documentation*. [Online]. Avail-

able: https://www.tensorflow.org/versions/master/how_tos/image_retraining/index.html

[10] “How to compile tensorflow with cUDA support on oSX,” *Medium @@fabmilo*. 2016 [Online]. Available: https://medium.com/@fabmilo/how-to-compile-tensorflow-with-cuda-support-on-osx-fd27108e27e1#.hc9eqipd6

[11] “Tensorflow android camera demo,” *GitHub repository*. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/android/README.md