Udacity Machine Learning NanoDegree Capstone Project: Product Image Classification

By James Nacino

Project Overview

In this Udacity course, I have been highly fascinated with image classification. Luckily for me I have found a project, hosted on Kaggle.com in partnership with Cdiscount, in which the goal is to classify images.

The company Cdiscount, is the largest non-food e-commerce company in France with over 30 million different products for sale. They want to be able to classify their product images to predict its respective product category. This is similar to how Amazon has a bunch of different categories for each of their products.

Currently Cdiscount classifies their products by analyzing the description of the products that the seller describes. Through a text classification algorithm, Cdiscount is able to classify the product into its respective product category (with some accuracy). However, now they want to go even further beyond, to improve their product classification accuracy. In order to do this, they are looking into image classification. Cdiscount wants to read the picture that the seller inputs to automatically classify the product into its correct category. Having the correct category for the product will make the experience easier for Cdiscount customers to find products from various sellers, and would make it easier for the sellers to acknowledge the correct category for the product they are selling.

NOTE: The writeup for this capstone goes directly with my Jupyter notebook. If you need to reference the code, please refer to that file; this writeup is organized very similar to the notebook. In the Juptyer notebook, I have comments in my code which for many points explains in more detail which decisions I made in the code.

Problem Statement

Cdiscount provided no ordinary image classification problem. I have been given a little over seven million images from Cdiscount which I will use to train my model, and they want to be able to classify their products correctly into one of 5270 different categories. In this problem, I will implement a convolutional neural network to classify the image's categories.

Solution 1 - The Solution that I backed away from, and my original thought to solving this problem

When I first started this problem, my goal was to implement a solution that would break this problem up into different sets of neural networks. I am going to explain my first thought process when I first went out to solve this problem.

I felt that 5270 different classes is a lot of labels to correctly predict which the image is associated with, which is why I thought of breaking this problem into multiple sets of neural networks. Cdiscount also provided a file which nests each of the 5270 categories into more general categories. So as a concrete example, a backpack and a purse would be in the same general category as provided by Cdiscount, but their specific category would be different. I was going to create a convolutional neural network which predicted the general category of the product. Then after running the products through my first neural network, I wanted to create many neural networks that are specific to the general categories.

So my machine learning pipeline would've looked like this:

1. Create and train a convolutional neural network which predicts the product's general category. There are 49 different general categories, so 49 different output nodes
2. Run each product through this neural network to predict its general category
3. Create and train new convolutional neural networks with products that have been predicted with a general category from the first neural network. This means that I will have 49 different neural networks that are specifically trained with products that within a general category. Each of these neural networks will have around 100 different output nodes. The logic for this is because 49 x 100 equals around 5000.

I decided not to follow up with this solution for many reasons, and will further explain why I ditched this solution later on when I test this out on a neural network in the solution statement section:

1. It was much more complicated to maintain and compile several neural networks and to train each one individually.
2. It would have taken longer to run this algorithm on the test data, which again was to predict the correct class of product using the first neural network, then the second neural network
3. The accuracy of the highest validation was around 55%, which is not as high as I would've hoped. This is because it still has to go through a second neural network, which will for sure bring the accuracy down when testing the final prediction

Solution 2 - Revised and implemented solution

This solution maintains the use of one convolutional neural network, but predicts the class of a product into one of 5270 specific product categories. The benefits of this is as follows:

1) It is much easier to maintain and implement one convolutional neural network
2) Testing the accuracy on the test dataset will be faster
3) Due to only having one convolutional neural network, it will be easier to add new layers, nodes, or a new structure in general to validate different results
4) Transfer learning (Inception model) to be used at the end to compare against the model that I built

Various Miscellaneous Problems That Will Be Addressed Later in this Report

1. Storing and pre-processing the data
2. Memory and Storage Issues
3. Setting up Google Cloud Computing
4. Setting up CUDA GPU Computing

Goals and Metrics for this project

Each Kaggle competition has a leaderboard. My goal entering my first Kaggle competition is to be in the top 50th percentile. This means that I want to achieve an accuracy score on the test dataset that is higher than the median participant's accuracy score. As I was doing this project, I did not know exactly what accuracy score would get me in the 50th percentile, however I had a general idea that the accuracy to get this ranking would be around 40 percent due to the current submissions.

Datasets and Inputs

Data Processing

In order to process the data, Cdiscount provided a 'train.bson' file which contained all of the image data, containing 7,069,896 images in total. This file is a mongoDB database data extract. It takes the form of a JSON-like structure. With this BSON file format, it wasn't easy to access images. In the Kaggle competition, they gave us starter code to read the BSON file. I had to further edit the code to store the data in the format that I wanted which I just easily wanted to access large batches of images. However, using the starter code was not what I initially wanted to do.

I initially tried many things to preprocess the data to where it could be properly loaded into a neural network. I first tried to convert the 'train.bson' file to a JSON object using a mongoDB tool called 'bsondump. I tried to do this because there were various Python modules where you can read in JSON files pretty easily. However, it turned out that the processed JSON file was in an incorrect file format for Python, and R programming languages to properly read in; something about converting the BSON file to JSON was done either incorrectly or corrupted the file.

So, I resulted to the code that they gave to read in the BSON file. However, the starter code that they gave us had no information on how to store the images. I wanted to store the images into an easily accessible file type that can store vasts amounts of data. I researched various file formats and saw that the 'H5' file format was able to store python arrays, and was able to handle large amounts of data efficiently.

The result was 140 different H5 files which stored the image arrays. Each of these H5 files contained 50000 images. I used separate H5 files as well to store the productID as well as the categoryID, however, these files did not nearly need as much memory since they were arrays that did not consist of image data but labels.

Processing the training dataset

I added a lot of things to the starter code that they gave us to process the BSON files. When I first started I ran into a problem. The problem was that the image arrays took a very long time to be appended together. The work around was to use pandas dataframes. I would temporarily use the pandas dataframes to append each image array in each cell under one column; this turned out to be a lot faster then appending two numpy arrays together. Then I converted the pandas column into a numpy array of image data. Each iteration of the loop stored 50,000 images and its respective category which was saved in H5 files. I was able to run this loop 40 times to get a training dataset size of 2,000,000 images. I would later split this training set into train and validation sets.

The H5 files were too large to fit into my computer's hard drive, in total the images consumed around 800 GB. To solve this, I had to buy an external hard drive which I could store all the image data into. I then had to figure how to connect my iPython notebook to an external storage device which I was able to do with Windows symbolic links.

Processing the testing dataset

The test dataset for this problem, they provided a file 'test.bson', consisted of 1,768,182 images. I did the same process here as in processing the training dataset by storing 50,000 images and the categoryId into multiple H5 files, with the last H5 file containing less than 50,000 images to evenly add up to the total amount of images.

Visualization - Pictures¶

Below are examples of the pictures that Cdiscount has provided for us. There are many different types of products, ranging from phone cases to batteries to CD albums.
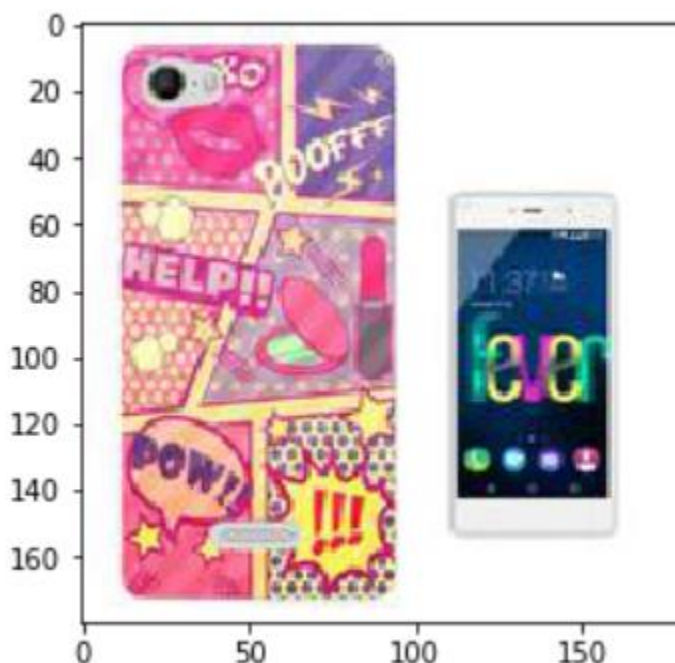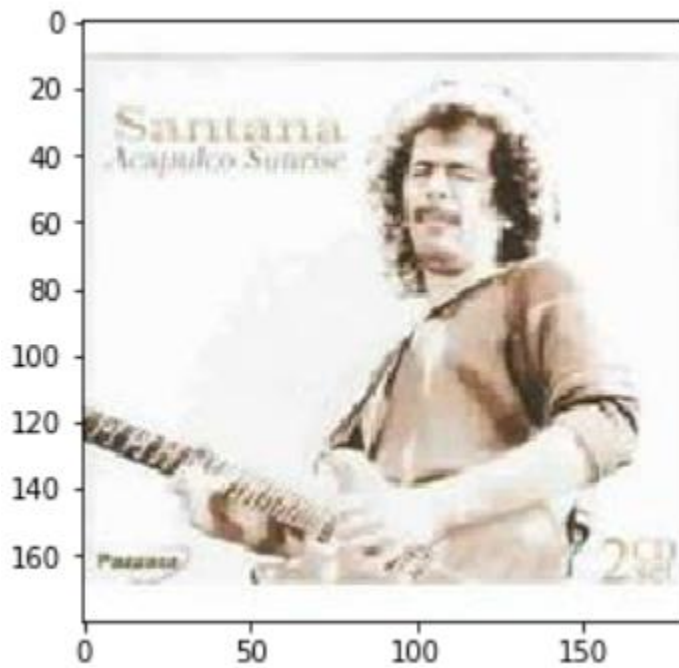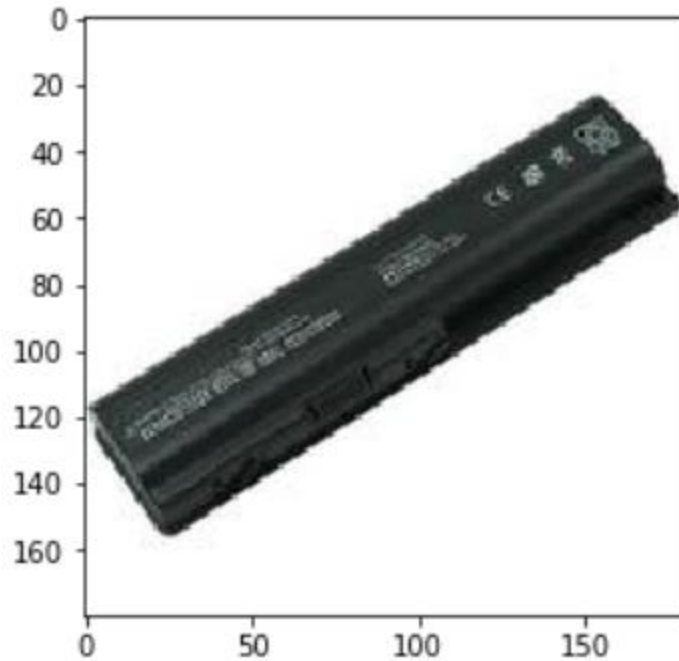
Categories

Cdiscount provided the 'category_names.csv' file shown below. This file contains a list of 5270 different 'category_ids'. However, I made modifications inside this file and used an excel function to append a new column, 'cat_1_encode'. This column basically encodes the 'category_level1' column into integer values ranging from 0-48. I have also appended a column 'catID_encode' which encodes the 'category_id'; so 'catID_encode' has integer values ranging from 0-5269. This is because there are

5270 different categories in total. I also appended a new column, 'cat_2_encode' which does the same to the 'category_level2' column (however I did not use this category level in my project). Encoding each of the columns is important because we are dealing with a classification problem, and we are predicting multiple classes.

	category_id	category_level1	category_level2	category_level3	cat_1_encode	catID_encode	cat_2_encode	cat_1_encodep2
0	1000021794	ABONNEMENT / SERVICES	CARTE PREPAYEE	CARTE PREPAYEE MULTIMEDIA	0	0	0	0
1	1000012764	AMENAGEMENT URBAIN - VOIRIE	AMENAGEMENT URBAIN	ABRI FUMEUR	1	1	1	0
2	1000012776	AMENAGEMENT URBAIN - VOIRIE	AMENAGEMENT URBAIN	ABRI VELO - ABRI MOTO	1	2	1	1
3	1000012768	AMENAGEMENT URBAIN - VOIRIE	AMENAGEMENT URBAIN	FONTAINE A EAU	1	3	1	2
4	1000012755	AMENAGEMENT URBAIN - VOIRIE	SIGNALETIQUE	PANNEAU D'INFORMATION EXTERIEUR	1	4	2	3

Creating the Training Set - One hot encoding 'cat_1_encode' column

In this project, I am consuming tons of memory when trying to divide my numpy arrays by 255. I needed to divide each pixel value in the array by 255 because for classification purposes it will produce more accurate results. I originally had a laptop that only had 8GB of RAM and that always ended up in a resource exhaust error because I did not have enough memory to divide the images.

I went to attempt to use the Google Cloud compute. I had a ubuntu virtual machine set up in the cloud. However, there were even further problems that arose from trying to use cloud computing. First, when I transferred files from my local workstation to the cloud, it took around three days to transfer just 300,000 images. This was most likely that my bandwidth was slow, but even with a better bandwidth, it still would have taken a while. Another problem with Google Cloud is that renting out a server with GPU compute capability cost over 1000 dollars a month which I did not even want to attempt setting up that workstation, even with the free 300 dollar credit that Google initially offers their new cloud users.

I decided to just buy a new laptop with more compute power (32 GB RAM) and a NVIDIA GPU; I probably would not have done this but I am starting an MS in Data Science soon and wanted to get a new laptop anyways. Even with a lot more memory, my new computer often froze when trying to divide the whole 50,000 image numpy arrays by 255.

I am dividing the image array by the value of 255 because it is the maximum pixel value and we need to get each value in the picture array to be a value between zero and one. Our predictions will be much more accurate when we have input values that are closer to one another.

I only used 1,600,000 of the images provided to fit my models due to the amount of time training. The remaining 400,000 images I used for my validation set; ultimately 80% of the images were used for training and 20% of the images that were processed were used for validation. These 2,000,000 images, I reformatted by dividing each picture's pixels by 255. If need be I will increase the amount of data I will use in this project to increase my accuracy.

With one hot encoding the 'cat_1_encode' column, what this did was that it created a numpy array of length 49 for the target labels; for example, the target label for a given image would consist of zeros, and a single value of a one where the label classification is true. I have also saved the fully processed one hot encoded 'cat_1_encode' labels into H5 files so that the array can be easily accessed later without taking up too much memory on my machine.

Creating Training Sets - One hot encoding 'catID_encode' column

I have also created the training sets for the 'catID_encode' column. This means that each of the pictures provided will need to be reformatted by dividing each pixel by 255, similar to what was done above. The difference here is that each of these pictures will be associated with its specific label that ranges from 0-5269, because there are 5270 unique category IDs. This is different than above, because this is the preprocessed dataset that I actually used to implement my final solution (which was the second solution listed in the 'Problem Statement' section.

With one-hot encoding I was able to create an array of zeros and ones which represented the true class of the image. For example, if the image's true classification was the 100th category, the associated array would show a '1' for the 100th index in the array and a '0' for the remaining 5269 indexes available in the array.

I saved 2,000,000 images and its respective category ID label in H5 files.

Solution Statement - Using Validation Accuracy as Evaluation Metric

In this section I have tested various convolutional neural network structures to determine which would obtain the highest accuracy score.

A convolutional neural network is one of the best ways to categorize image data in today's world. The reason for this is because of its success in image training competitions such as ImageNet. Another benefit of convolutional neural networks is that you can combine nodes together from one layer to the next to create even more complex non-linear classifiers. With 5270 different images in the dataset, it is probably safe to say that we would need a complex classifier that can distinguish many unique patterns within various images.

CPU and GPU summary

This is the information on the CPU and GPU that I'm using. I am using a NVIDIA GTX 1070 GPU and have set it up correctly to work with Tensorflow. The GPU on my computer will make it a lot faster to train the convolutional neural network.

```
}
incarnation: 12345590033269605352
physical_device_desc: "device: 0, name: GeForce GTX 1070, pci bus
id: 0000:01:00.0, compute capability: 6.1"
]
```

I have 32GB of RAM/memory on my machine. I only had 31GB available to use to implement various neural networks.

```
import psutil
psutil.virtual_memory()
```

```
svmem(total=34314883072, available=31125389312, percent=9.3, used=
3189493760, free=31125389312)
```

Project Design & Benchmark Model

This next section is very long. Listed is a summary of what is happening in the next couple pages when trying to figure out what was the best solution to implement. And my thought process when running each experiment.

Also, my benchmark model is going to be the first three experiments of when I when I went to test solution two. I should be able to improve upon these first three experiments, which is why it is my benchmark model. However, my initial goal was to use logistic regression as a benchmark model but I wasn't able to implement it with the large dataset.

1) Testing Solution 1
   a) Experiment 1
      i) **Summary**: Implemented a neural network which loaded 50,000 images at a time and trained batch sizes of 20 images. Classifying the image into a class of 0-49. Saving the model weights after each of the 50,000 images have been fitted on the neural network. Then loading in the next 50,000 images, and loading the previous weights, train the images, and save the weights again, continuing this process on 300,000 images.
      ii) **General thoughts**: There had to be a better, more efficient way to load in the images to train the neural network rather than loading the model weights after each iteration of the loop.
   b) Experiment 2
      i) **Summary**: This time used Keras's 'fit_generator function to train the neural network, using single epochs that consisted of 300,000 images. Implemented a neural network which loaded 50,000 images at a time and trained batch sizes of 20 images. Trained using 10 epochs

ii) **General thoughts**: Using 'fit_generator' was a lot more efficient to load in the data and train a neural network on a large dataset.

2) Testing Solution 2
   a) Experiment 1
      i) **Summary:** Used the Rmsprop optimizer to run on a custom built neural network to train on 800,000 images and validate on 200,000.
      ii) **General thoughts:** The accuracy here was very low
   b) Experiment 2 – Benchmark model
      i) **Summary:** Used the Adam optimizer to run on a custom built neural network to train on 800,000 images and validate on 200,000
      ii) **General Thoughts:** The accuracy increased a large amount from experiment 1. This ended up returning the highest validation accuracy from my first three experiments, so I will use this as my benchmark model
   c) Experiment 3
      i) **Summary:** Used the Adadelta optimizer to run on a custom built neural network to train on 800,000 images and validate on 200,000
      ii) **General Thoughts:** The accuracy decrease from experiment 2, but much higher accuracy than experiment 1
   d) Experiment 4
      i) **Summary:** Created a new neural network structure, using the Adam optimizer to train on 800,000 images and validate on 200,000
      ii) **General Thoughts:** The accuracy decreased from experiment 2
   e) Experiment 5
      i) **Summary:** Used transfer learning (Inception model), using the Adam optimizer to train on 800,000 images and validate on 200,000
      ii) **General Thoughts:** The accuracy increased from experiment 2.
   f) Experiment 6
      i) **Summary:** Used transfer learning (Inception model), using the Adam optimizer to train on 1,600,000 images and validate on 400,000
      ii) **General Thoughts:** The accuracy increased from experiment 5 and is the highest I obtained throughout the project

Testing Solution 1

Experiment 1 - Based on Solution 1 (which is the solution I did not move forward with)

I am going to use the following structure for this neural network, this is the first neural network in which the goal is to classify each of the images into its general category that includes 49 different classes.

We want to specifically declare the Relu activation function because it mitigates the vanishing gradient problem. This is because the gradient of the Relu activation function is one if the output is greater than zero, it is much easier for the backpropogation step to more quickly find a local or global minimum which minimizes the error function in our model.

Below is my model summary:

```
Using TensorFlow backend.
```

```
_____
Layer (type)                     Output Shape              Param #
===============================================================
conv2d_1 (Conv2D)                (None, 89, 89, 80)         3920
_____
dropout_1 (Dropout)              (None, 89, 89, 80)         0
_____
max_pooling2d_1 (MaxPooling2     (None, 44, 44, 80)         0
_____
conv2d_2 (Conv2D)                (None, 43, 43, 60)         19260
_____
dropout_2 (Dropout)              (None, 43, 43, 60)         0
_____
max_pooling2d_2 (MaxPooling2     (None, 21, 21, 60)         0
_____
dense_1 (Dense)                  (None, 21, 21, 40)         2440
_____
dense_2 (Dense)                  (None, 21, 21, 20)         820
_____
flatten_1 (Flatten)              (None, 8820)               0
_____
dense_3 (Dense)                  (None, 49)                 432229
===============================================================
Total params: 458,669
Trainable params: 458,669
Non-trainable params: 0
```

Training my model¶

To train my model, I used a for loop to read in each of the H5 files. With the image array data, I fed that into my neural network to train it. So 50,000 images were fed into the neural network for one iteration of the for loop, then the next 50000, and so on. I loaded the model weights from the previous epoch so that my neural network can continue training, but on a different set of the next 50000 images. I also specified a callback so that the best model weights can be saved. I only used

300,000 images to train this model as I wasn't initially sure this was how to correctly feed inputs into a neural network if you had multiple image files.

I found that I needed a GPU because it took a very long time for me to train my neural network using just a CPU. I was also only able to train my model with 20 images at a time (batch size) due to an error I received that says that my system was exhausted of memory resources; I initially wanted to train my model with 1,000 images at a time.

Results – Experiment 1

```
Train on 40000 samples, validate on 10000 samples
Epoch 1/1
39980/40000 [==============================>.] - ETA: 0s - loss: 1.9115
- acc: 0.5402Epoch 00000: val_loss improved from inf to 2.10149, savin
g model to weights_best.hdf5
40000/40000 [==============================] - 844s - loss: 1.9116 - a
cc: 0.5402 - val_loss: 2.1015 - val_acc: 0.5756
Train on 40000 samples, validate on 10000 samples
Epoch 1/1
39980/40000 [==============================>.] - ETA: 0s - loss: 2.5832
- acc: 0.3566- ETA: 6s - loss: 2Epoch 00000: val_loss did not improve
40000/40000 [==============================] - 695s - loss: 2.5835 - a
cc: 0.3565 - val_loss: 2.7337 - val_acc: 0.3516
Train on 40000 samples, validate on 10000 samples
Epoch 1/1
39980/40000 [==============================>.] - ETA: 0s - loss: 2.5850
- acc: 0.3415Epoch 00000: val_loss did not improve
40000/40000 [==============================] - 938s - loss: 2.5848 - a
cc: 0.3416 - val_loss: 2.4990 - val_acc: 0.3504
Train on 40000 samples, validate on 10000 samples
Epoch 1/1
39980/40000 [==============================>.] - ETA: 0s - loss: 2.5837
- acc: 0.3382Epoch 00000: val_loss did not improve
40000/40000 [==============================] - 993s - loss: 2.5838 - a
cc: 0.3382 - val_loss: 2.6028 - val_acc: 0.3131
Train on 40000 samples, validate on 10000 samples
```

It appeared that from each of the six epochs (not all epochs shown above), it was the first epoch that was able to reduce the most error in the model and obtain the highest accuracy, with a validation accuracy of 57.6%. This is most likely because not all of the categories were included within the first 50,000 images.

However, training a model like this seemed very inefficient. This is because a single epoch is supposed to consist of the entire training dataset, and here an epoch consisted of a different batch of 50,000 images. I initially thought that just by loading the model weights from the previous batch

of 50,000 images to train on the next 50,000 images would do the trick, but it clearly wasn't a viable solution.

To reiterate, the goal of solution one was to be able to classify an image into 49 general categories instead of 5270 unique classes. What I was imagining was that I would run the image through the fitted neural network model which will predict into one of 49 different general categories. Then from there, the image would be transferred to another specific fitted neural network that has all the specific unique classes for the general category.

Experiment 2 - Based on Solution 1, Training using fit_generator

After doing some research it appeared that the best way to read in multiple image files for training under one epoch was to use the 'fit_generator' function in Keras. With this function I was able to loop in all of the images under a single epoch to train. This was a lot more efficient than what I performed for experiment 1. Below is my model summary for experiment 2.

```
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 177, 177, 100)     4900

dropout_1 (Dropout)          (None, 177, 177, 100)     0

max_pooling2d_1 (MaxPooling2 (None, 88, 88, 100)       0

conv2d_2 (Conv2D)            (None, 87, 87, 80)        32080

dropout_2 (Dropout)          (None, 87, 87, 80)        0

max_pooling2d_2 (MaxPooling2 (None, 43, 43, 80)        0

conv2d_3 (Conv2D)            (None, 42, 42, 60)        19260

dropout_3 (Dropout)          (None, 42, 42, 60)        0

max_pooling2d_3 (MaxPooling2 (None, 21, 21, 60)        0

dense_1 (Dense)              (None, 21, 21, 40)        2440

dense_2 (Dense)              (None, 21, 21, 20)        820

flatten_1 (Flatten)          (None, 8820)              0

dense_3 (Dense)              (None, 49)                432229
=================================================================
Total params: 491,729
Trainable params: 491,729
Non-trainable params: 0
```

Results – Experiment 2

```
Epoch 6/10
11999/12000 [==============================>.] - ETA: 0s - loss: 1.3376
- acc: 0.6327Epoch 00005: val_loss improved from 1.77329 to 1.75527, s
aving model to weights_bestID1Generator.hdf5
12000/12000 [==============================] - 4998s - loss: 1.3376 -
acc: 0.6327 - val_loss: 1.7553 - val_acc: 0.5542
```

The highest accuracy that I received when training my model was on the sixth epoch where I received a validation accuracy of 55%.

I will not continue with the same solution structure due to the results I received here. I believe that since the general product categories are so broad, some products may share patterns with others products in different general categories and the accuracy is very low.

Due to a low accuracy and the training time that it will take to train many different neural networks, I will go ahead and see if I can just implement one convolutional neural network that directly predicts the product into its specific class that includes 5270 different classes.

Testing Solution 2 – The final solution I implemented to solve the problem

Now I am going to directly predict images into its specific class that includes 5270 different classes. Predicting directly into the specific category will save a lot of training time and make things simpler to manage. It will also make it much easier to experiment with the neural network's structure by adding any layers, filters, or nodes to improve the accuracy. I am implementing the 'fit_generator' function in all of the experiments for solution 2.

For the first five experiments I will train on 800,000 images and validate on 200,000 images. I expanded the number of images to train on because with a larger class size of 5270 different classes, I needed to make sure that there were enough samples for the neural network to distinguish unique patterns amongst all the classes.

The first three experiments will have the same neural network structure (same number of nodes, layers, etc.), but with different optimizers to minimize error. The first experiment I will try Rmsprop, the second Adam, and the third will be Adadelta.

The fourth and fifth experiments will be about changing the structure of the neural network, but using the optimizer that achieved the highest validation accuracy from the first three experiments. The fourth experiment, I will restructure a custom model on how I see fit, then with the fifth experiment, I will use transfer learning to train the Inception model with the data.

Below is the model summary that was used for the first three experiments. I used batch sizes of 20.

```
Layer (type)                     Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)                (None, 177, 177, 100)     4900

dropout_4 (Dropout)              (None, 177, 177, 100)     0

max_pooling2d_4 (MaxPooling2     (None, 88, 88, 100)       0

conv2d_5 (Conv2D)                (None, 87, 87, 80)        32080

dropout_5 (Dropout)              (None, 87, 87, 80)        0

max_pooling2d_5 (MaxPooling2     (None, 43, 43, 80)        0

conv2d_6 (Conv2D)                (None, 42, 42, 60)        19260

dropout_6 (Dropout)              (None, 42, 42, 60)        0

max_pooling2d_6 (MaxPooling2     (None, 21, 21, 60)        0

dense_4 (Dense)                  (None, 21, 21, 40)        2440

dense_5 (Dense)                  (None, 21, 21, 20)        820

flatten_2 (Flatten)              (None, 8820)              0

dense_6 (Dense)                  (None, 5270)              46486670
=================================================================
Total params: 46,546,170
Trainable params: 46,546,170
Non-trainable params: 0
```

Experiment 1 Results – Training using Rmsprop optimizer

```
Epoch 1/18
39999/40000 [=============================>.] - ETA: 0s - loss: 6.7121
 - acc: 0.1531Epoch 00000: val_loss improved from inf to 7.57415, savin
g model to weights_bestcatIDGenerator.hdf5
```

The first epoch, returned the highest validation accuracy of 15.3%. Although it says I ran this experiment for 18 epochs, I stopped it after 8 because the validation loss kept on increasing; meaning the validation accuracy kept on declining.

Experiment 2 Results – Training using Adam optimizer

Due to the first experiment and the training time it took, I felt that it was best to keep this to 10 epochs. And I thought 10 epochs would be enough to minimize the loss in the error function.

```
Epoch 4/10
39999/40000 [=============================>.] - ETA: 0s - loss: 3.7453
- acc: 0.3878Epoch 00003: val_loss improved from 5.90290 to 5.87724, s
aving model to weights_bestcatIDGeneratorAdam.hdf5
40000/40000 [==============================] - 11238s - loss: 3.7453 -
acc: 0.3878 - val_loss: 5.8772 - val_acc: 0.2424
```

Experiment 2 saw a significant increase in accuracy as it jumped to 24.2% accuracy. I believe this is because the Adam optimizer uses momentum when trying to minimize the error function. What momentum for the Adam optimizer does is that it uses a more complicated exponential decay for the learning rate that consists of not just considering the average (first moment), but also the variance (second moment) of the previous steps. Each step is performed when a batch of 20 images have backpropagated through our neural network.

Experiment 3 Results – Training using Adadelta optimizer

```
Epoch 1/10
39999/40000 [=============================>.] - ETA: 0s - loss: 4.8301
- acc: 0.3208Epoch 00000: val_loss improved from inf to 6.12848, savin
g model to weights_bestcatIDGeneratorAdaDelta.hdf5
40000/40000 [==============================] - 11845s - loss: 4.8301 -
acc: 0.3208 - val_loss: 6.1285 - val_acc: 0.2045
```

The first epoch managed to get the highest validation accuracy of 20.5%.

Experiment 4 Results – Changing the neural network structure and training using the Adam optimizer

Since the highest validation accuracy received came from using the Adam optimizer, I will use that optimizer for experiment 4.

Based on the verbose results of experiment 2 (which used the Adam optimizer), it appeared that the neural network overfit the data. This was because the highest accuracy on the training set achieved was 41%, on the ninth epoch, but the validation accuracy was at 23.6%.

Because it seemed like my model was overfitting the data, I reconstructed the model for experiment 4 to make the model simpler which meant decreasing the amount of layers and nodes. Below is the model summary for experiment 4.

```
Layer (type)                   Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)              (None, 177, 177, 64)      3136

dropout_1 (Dropout)            (None, 177, 177, 64)      0

max_pooling2d_1 (MaxPooling2   (None, 88, 88, 64)        0

conv2d_2 (Conv2D)              (None, 87, 87, 32)        8224

dropout_2 (Dropout)            (None, 87, 87, 32)        0

max_pooling2d_2 (MaxPooling2   (None, 43, 43, 32)        0

dense_1 (Dense)                (None, 43, 43, 20)        660

dense_2 (Dense)                (None, 43, 43, 10)        210

flatten_1 (Flatten)            (None, 18490)             0

dense_3 (Dense)                (None, 5270)              97447570
=================================================================
Total params: 97,459,800
Trainable params: 97,459,800
Non-trainable params: 0
```

The results for experiment 4 below also show that the model was still overfitting. This was also the best validation accuracy received from the 10 epochs which was four percentage points lower then the results from experiment 2.

```
Epoch 5/10
39999/40000 [============================>.] - ETA: 0s - loss: 4.225
5 - acc: 0.3505Epoch 00004: val_loss improved from 6.51178 to 6.4501
2, saving model to weights_bestcatIDGeneratorAdamV2.hdf5
40000/40000 [=============================] - 11186s - loss: 4.2254
- acc: 0.3505 - val_loss: 6.4501 - val_acc: 0.2069
```

Experiment 5 Results – Using the Inception Model to train the data (transfer learning)

Using transfer learning it appeared that the validation accuracy improved by a little over 5 percentage points from experiment 2! I used the code found here https://keras.io/applications/ to implement the inception model with my own dataset.

```
Epoch 8/8
39999/40000 [=============================>.] - ETA: 0s - loss: 3.6347
- acc: 0.3989Epoch 00007: val_loss improved from 5.15022 to 5.13357, s
aving model to weights_bestcatIDGeneratorInception.hdf5
40000/40000 [==============================] - 9576s - loss: 3.6347 -
acc: 0.3989 - val_loss: 5.1336 - val_acc: 0.2987
```

Experiment 6 Results – Inception Model with more data!

It has been said many times throughout the Udacity course that the more data, the more accurate the prediction. Since the Inception model provided the most accurate model for predicting the classes of these products, I am going to double the size of my training data from 800,000 to 1,600,000 images. My validation data size will also be doubled from 200,000 to 400,000 images.

```
Epoch 4/10
79999/80000 [=============================>.] - ETA: 0s - loss: 2.9087
- acc: 0.4789Epoch 00004: val_loss improved from 3.38398 to 3.38299, s
aving model to weights_bestcatIDGeneratorInceptionV2.hdf5
80000/80000 [==============================] - 19102s 239ms/step - los
s: 2.9087 - acc: 0.4789 - val_loss: 3.3830 - val_acc: 0.4671
```

By simply doubling my training set, the validation accuracy increased by around 16% points from 29.87% from the last experiment to 46.71%!!!

I finished this last experiment, experiment 6, the day right before the Kaggle competition was due. I wish that I had more time, as I would go back and preprocess as much of the entire dataset as possible to train the Inception model.

Final Results

Model Evaluation

With all of the models that I experimented with, I used the transfer learning model from experiment six to use as my final model for submitting predictions as it returned the highest validation accuracy.

With the 'Test.bson' file that Cdiscount provided, 1,768,182 images, I used the images in that file to feed into my trained model from experiment six. Once my model returned a prediction for all the pictures in that file I saved the predictions into a CSV file which was submitted to the competition.

307	—	Vladislav Kassym		0.46290	5	1mo
308	—	JamesNacino		0.46234	3	4d

My initial goal at the beginning of this project was to at least place in the 50th percentile for all competitors who submitted their predictions. At the end of the competition I was ranked 308/627, so I managed to meet this goal.

Cdiscount's Image Classification Challenge
Categorize e-commerce photos
Featured · 3 days ago · 🏷 multiclass classification

Cdiscount

$35,000
308/627

When comparing the validation accuracy returned from experiment six and the test accuracy when submitting this project on Kaggle.com, the validation accuracy was higher than the test accuracy by .48% (46.71 – 46.23). Due to the small difference, I can safely assume that the validation set that I used was similar to the test set.

Justification

The final results that I submitted to Kaggle were stronger than the benchmark result that I initially reported. This is because my benchmark model obtained a test accuracy of 24.69% and my final model obtained a test accuracy of 46.23%. As shown below, I also submitted the predictions on the test set from my benchmark model from experiment 2.

submissionsv2Adam.csv	0.46234
4 days ago by JamesNacino	
add submission details	

submissionsv1Adam.csv	0.24686
16 days ago by JamesNacino	
add submission details	

Conclusion

Reflection

This was a project that focused a lot on data preprocessing and training various CNN structures. Listed below is a summary of what was implemented to reach my final solution and the problems that arose at each of these various phases of the project.

1) Data preprocessing the BSON files
   a) I had to process the training dataset that Cdiscount provided into a more accessible format
      i) I initially tried using a mongoDB tool call bsondump to convert the BSON file into a readable JSON file that I can load into my Python environment. However, I couldn't properly do this
      ii) I ended up editing the starter code that the competition hosts gave us to process the BSON files to save the image files and their corresponding class data in H5 files. It

took a couple days for all 7 million images to be saved into H5 files (each H5 files stored a batch of 50,000 images)

b) I had to reformat the data needed by dividing each pixel by 255 and one hot encode the output labels to properly train the CNN

 i) I initially tried to divide the image arrays by 255. Each image array consisted of 50000 images. Dividing each of these 50000 images' pixels by 255 froze my laptop

 ii) I then set up a Google Cloud compute environment where I wanted to finish processing my data and train my models. However, it took three days just to upload 300,000 images to the cloud. With this problem and the fact that I couldn't rent out a GPU configured server made me ditch the cloud computing environment

 iii) I bought a new laptop with 32GB RAM and an NVIDIA GPU. With this new laptop it was possible to reformat each of the arrays stored in the H5 files properly needed to train my models (however, many times my laptop would still freeze from memory resource limitations). Later on in this project near the end I found the most efficient way to divide each image by 255, but it was too late as the project was nearing its deadline

2) Training my model

a) Setting up the NVIDA GPU on my machine for GPU computing

 i) Had to watch a series of videos and read tutorial how to properly set up the NVIDA GPU with Tensorflow

b) Constructing various CNN structures to fit the training data and analyzing which model would yield the highest validation accuracy

Improvement

If I were to start this project from scratch, I would change a lot of things to get me started on training the models much earlier on. If I were able to focus more on training the data, than data processing I believe that I would have been able to get a higher accuracy score.

Probably the biggest problem that I had was data processing and how to get enough data to work with. It was a big trouble just to get 300,000 formatted images to feed into my neural network. However, incrementally as this project moved forward I tried various ways of formatting data. I was able to train many of my experiments with 800,000 images and train with 200,000 but I had to change my code up a lot to save the memory resources on my machine to efficiently process this data. Near the end of the project, I was able to efficiently change the code where my computer wouldn't freeze when I tried to reformat the images. In the future, I would save the images on the H5 files in batches of 20,000 instead of 50,000 as that should prevent any resource exhaust error.

It took me a long time to get 1,000,000 fully processed and reformatted images. However, as the project neared the deadline, I had a breakthrough and was able to change up my code and process another 1,000,000 images faster. Unfortunately, that was near the Kaggle competition deadline and I would have liked to have trained my model with all 7+ million images that they provided in the 'train.bson' file.

Another thing that I would like to improve upon is my selection of model training. I would like to go straight to transfer learning next time I would do a project like this, and change the model to further increase the accuracy. Building my own models was great experience, but if I went straight to transfer learning I could have used that as my benchmark model. This would have been a good idea because these transfer learning models are already proven image classifiers.

As mentioned before, I also would have liked to train with more data as I know my accuracy score would have increased if I trained with the entire dataset.