Training Xception model for Kaggle competition “Cdiscount’s Image Classification Challenge”

Ardi Loot

Institute of Physics, University of Tartu, W. Ostwald St. 1, 50411 Tartu, Estonia

(Dated: January 5, 2018)

This commutation is about training the Xception model for the Kaggle competition “Cdiscount’s Image Classification Challenge”. The paper will briefly describe all methods/code (github.com/ardiloot/CDiscountClassifier) used to train the model for best classification performance. Mainly, the effect of the augmentation (both train and test time) and algebraic ensemble methods were studied. In the competition, the model without augmentation, no test-time-augmentation and with power law ensemble method scored the best (accuracy 0.72582) and guaranteed the 64th position out of 627 teams.

I. INTRODUCTION

The aim of this work was to participate in the Kaggle competition “Cdiscount’s Image Classification Challenge” (https://www.kaggle.com/c/cdiscount-image-classification-challenge). Cdiscount is a large and continuously growing online non-food department-store and as the number of products is huge (around 30 million), the classification of the products is a hard task. The goal of the competition was to develop an algorithm that is able to automatically classify products by their images into 5720 existing categories. In other words, the competition is essentially standard image classification task, where neural networks have been huge success lately.

On the other hand, several details make this competition more difficult than a standard image classification. First, the number of categories (5720) is really huge as standard ImageNet competition only has classification into 1000 categories. Secondly, the training data is really big (58.2 GB, 7 million products, each having 1-4 images). Thirdly, the dataset contains a lot of difficult or even wrong categories (e.g. different styles of books and music). And finally, the variable number of images (1-4) for each product actually makes the classification task non-standard and the predictions of different images must be combined.

For the training, the images around 7 million products (each having unique ID) were made available with information about the correct category ID. Each product could contain from one to four images with size 180×180. All the training data was combined into single BSON file with size 58.2 GB. Similarly, the images for around 1.8 million products without correct category ID was given (14.5 GB) for testing.

In addition to the image data, also the hierarchical structure of the categories were made available in CSV file. Each category were described by corresponding level1, level2 and level3 name in French. As it was pointed out, also in the file descriptions, the category information is not necessary and it is totally up to the competitors to use this information or not.

As a part of this paper Xception model was trained for this classification task. The methods and models are described in Sec. II, the results are given in Sec. III and in the end the results are concluded.

II. METHODS AND MODELS

A. Reading train/test data

The first technical problem was to read huge training and testing datasets. In this case, the compressed images were over 58.2 GB in size and it corresponds to around 1 TB of memory in case of uncompressed images. It is clear, that only a small part of the images could be in the RAM at the same time. The complexity is also added by the fact, that in order to efficiently train the neural network, the training data must be provided in random batches. It means, that sequential reading of the training data from beginning to end is not going to fully solve the problem.

To solve this problem several kernels were posted to the competition’s webpage. My strategy to read the training and test data closely resembles the solution proposed by user “Human Analog” in kernel “Keras generator for reading directly from BSON”[1], however, I had to develop my own considerable different version for several reason. Here only a short idea of the algorithm to handle big data will be presented.

First, the provided BSON files consists of documents (one per product), that usually are read from the beginning to end in a sequential fashion. However, it is not a strict requirement, as different documents could be decoded independently. The only information necessary for that is the knowledge of the location of the binary data in the BSON file. In other words, only the seek position and the length of binary data is required for reading random BSON document (i.e. random product). This, however, is easily possible to precalculate. It requires one full sequential read through BSON file to mark up the seek position of each product binary data and its length.
In case of training data, the products must also be split into train and validation sets. The choice in this work was to leave 10% of the train products for validation. Extra care was taken to keep this ratio in every category, as the categories had really different sizes[2]. Smaller than usual validation split size (10% instead of 20%) is justified by a very large training dataset.

As not all of the training images are in memory at the same time, it was not possible to use ordinary fit method of the Keras model. Instead, method fit_generator was necessary. As a input, it requires generator which returns the batch data on the call of next() method. In Kaggle community, the proposed method was to extend Iterator class from keras.preprocessing.image. In our case, it did not work well, because we wanted to used artificial epoch size containing 2 million pictures (instead of full epoch containing around 10 million images). Such definition of artificial epoch size is beneficial if using model callbacks, because e.g. Tensorboard and ModelCheckpoint callbacks only save information in the end of the epoch. However, in this case the training for one full epoch took considerable amount of time (up to 24h) and it made the on_epoch_end() callbacks too infrequent for practical use. The Keras fit_generator supports parameter steps_per_epoch, but at least in case of Keras 2.1.1 it was ignored in the case of keras.preprocessing.image.Iterator. The behavior of this parameter was changed/fixed only recently in Keras 2.1.2. Our solution (see github.com/ardiloot/CDiscountClassifier) was to write custom iterator to support epochs with custom size and provides thread-safe reading of random batches from BSON file.

The augmentation of the images were done in a standard way by using ImageDataGenerator class from keras.preprocessing.image. Additionally upscaling and random/center crop functionality were added.

In terms of performance, it is very important to use SSD instead of HDD drives. As the reading from BSON file is random, the performance in case of HDD is very poor and it will be the main bottleneck. Happily the GPU nodes of the cluster of University of Tartu has 5 TB local SSD storage.

B. Selection of model

Out of the box Keras supports following pretrained (ImageNet) models: Xception, VGG16, VGG19, ResNet50, InceptionV3, InceptionResNetV2, MobileNet.[3] The Xception model were selected out of this list because it had very good (second in goodness) top-1 accuracy (0.790) for the small number of parameters (ca 22 million). The only model with slightly better top-1 accuracy in this list were InceptionResNetV2 (0.804), however it also has more than two times more parameters. The low number of parameters is important for the training speed and also because of the memory restrictions on GPUs. Also, Xception model were reported to provide very competitive results by other competitors.[4]

C. Xception

Xception was proposed by F. Chollet, the creator and the chief maintainer of the Keras library in 2016 [5]. The author sees Xception model as an extreme version of Inception model with maximally large number of towers. The Inception architecture was first proposed by C. Szegedy, et al. in 2014 as GoogLeNet [6]. Inception type models (V1, V2, V3, Inception-Resnet) have proved to be one of the best performing models on the ImageNet dataset [5, 7]. The fundamental building block of Inception-type model is Inception module (see Fig. 1). It consists of four towers, one is simple 1 × 1 convolution layer, then there are towers with 3 × 3 and 5 × 5 convolution following 1 × 1 convolution, and finally average pool tower. In the end the results are concatenated.

Figure 1. Example of Inception module.

One possible explanation is provided on hacktill-dawn.com blog.[8] It is claimed, that in usual convolutional neural networks it is really hard design choice to select the best filter size and appropriate amount of pooling layers. Inception model solves this problem by including all of them in every layer. The additional 1x1 convolutions before final convolution is just for down-sampling the channel space to keep the amount of operations under control.

Another angle of view is presented in the Xception paper [5]. It is pointed out, that usual convolutional neural network tries to learn the filters in 3D space. It means, that is tries to map the correlations simultaneously in the spatial and the channel domains. The idea of Inception model is, that correlations in the spatial and in the channel domain are addressed separately. First, 1 × 1 convolution maps correlations in the channel domain and then larger filter maps the correlations in the spatial domain. Such decoupling enables to learn richer features more easily and with less parameters.

Xception model is kind of extreme version of very successful Inception model. It maximizes the number of tow-
ers in Fig. 1 keeping the number of $1 \times 1$ filters equal to one. In other words, Xception module relies on $1 \times 1$ convolution to map cross-channel correlation and then separately map the spatial correlation of every output channel.

The schematic representation of Xception model is shown in Fig. 2. The main building-blocks are Xception modules (ReLU + SeparableConv) and the residual connections around them. The model supports images larger than $70 \times 70$ in size. In the case of classification task the final 2048-dimension vector is followed by the dropout (optional) and the dense classification layer.

![Schematic representation of Xception model](image)

**Figure 2.** The schematic representation of Xception model. The main building-blocks are Xception modules (ReLU + SeparableConv) and the residual connections around them. [5]

### D. Optimization

In the original Xception paper ordinary stochastic gradient descent (SGD) with momentum were used. It is a quite usual choice of optimizer for training large models and it was popular choice also among this competition (on the basis of Kaggle discussions). Another popular optimizer (also used in this work) is Adam [9]. In this work, Adam was preferred because of superior speed of convergence in a few experimental tests to fine-tune Xception model with precalculated weights (based on ImageNet). Only the learning rate parameter were adjusted to $5 \cdot 10^{-3}$ and will be discussed further in the results section.

Keras has builtin Adam optimizer, however it unfortunately does not support gradient accumulation. It becomes a problem because of a limited amount of GPU memory. For example, in the case of 12GB GPU memory, the maximum batch size were around 90 in the case of $180 \times 180$ images. However, it is reported by many Kaggle users, that they use batch size around $512 - 1024$ for good result[10].

One solution to increase the batch size is to use multiple GPUs. Here, it was done with *multi_gpu_model* method from *keras.utils*. Essentially, it splits larger batch between GPUs, does calculations and then combines the results on CPU. Such solution works well and is fast, however for this work, only up to 2 GPUs were realistically available, so only batch size up to 180 was possible.

To use even larger batch sizes, the Adam optimizer class were modified to support gradient accumulation. The idea is, that instead of updating parameters after every batch, we accumulate the gradients of the batches and make updates only after summing for example 10 batches. Such algorithm effectively increases the batch size to any number. Unfortunately, Keras does not support gradient accumulation out of the box and had to be implemented separately. The extended Adam optimizer is based on the idea mentioned in Keras GitHub[11].

The main models in this work were trained on two GPUs, batch size 128 and number of accumulations 12. Effectively it results in 64 samples per GPU and in total effective batch size $12 \cdot 128 = 1536$. 
E. Regularization

Two sources of regularization were used in this study: dropout and data augmentation. In the case of dropout, only small rate 0.2 were used because of quite big dataset. As data augmentation ImageDataGenerator from keras.preprocessing.image were used along with custom cropping. Two main configurations were:

1. no-augmentation: No regularization at all
2. augmentation: Random crop (bicubic upscale to 200 × 200 followed by random crop 180 × 180) with probability 0.75 and horizontal flip with probability 0.5.

F. Predictions

As each product may have up to four images, the procedure of making predictions by trained model is not a straight-forward task. As we have multiple images per product, we have to adapt some kind of rules/algorithm to combine predictions from multiple images to make a single category prediction. Many different ways exists to solve this problem. For example, larger neural network may be formed that already takes four images as input. Similar effect could be achieved, if the activations for each image is saved and post-process afterwards by neural network or any other classifier. However, saving all the activations for post-processing would take approximately up to 40 GB of space and additional computational resources for post-processing. As a simple alternative algebraic ensemble methods were used. Probably the simplest is the arithmetic mean of the activations: the mean over activations of every category. Among arithmetic mean, also other ways to combine the activations of different images were tested: product, root-mean-square (RMS), median, mean of activations in power 0.1, maximum activations and as a baseline using only activations of the first image.

Another method to improve predictions accuracy is to use test-time-augmentation (TTA): calculate the activations for a single image multiple times by different augmentation each time. It has been reported to work really well by increasing the top-1 accuracy around 2% by using multiple crops [12]. In this work, also TTA was attempted by making predictions of five augmented images (same augmentation as in the case of training).

III. RESULTS

![Training accuracy curves](image)

Figure 3. The training accuracy curves for two main models: with augmentation (augmentation) and without (no-augmentation). The accuracy calculated on the training data is given by solid line and the accuracy of the validation is given by solid dots. The dashed red vertical lines represent the timing of the learning rate decay.

Many training attempts were made to develop training scripts and to optimize model hyper-parameters. The first attempts were made by down-scaled images (90×90) and only the dense layer were fine tuned (otherwise pre-calculated weights were used). Such model did not work well and the accuracy only up to 0.4 were achieved. The results slightly improved, if original 180 × 180 images were used. To achieve better results, more parameters...
were made trainable. If in addition to the dense layer, also last four blocks (determined by residual connections in Fig. 2) the results improved up to 0.66, which is much better, but still not enough to even guarantee the position in TOP100. Following this argumentation, it was clear, that training the full network was necessary. In this section, we first describe the exact procedure for training and discuss the features of the training curve. Secondly, the results of different methods to ensemble the predictions are compared.

A. The training of the full network

As described in Sec. IIE, two different strategies for regularization were used: augmentation (small dropout, random crop, horizontal flip) and no-augmentation (no regularization at all). The training accuracy curves are displayed in Fig. 3. The solid lines represent the mean accuracy of \(20 \cdot 10^3\) images and the solid circles represent the validation over random \(50 \cdot 10^3\) images. Only a small amount of total validation set were used in this place to speed up the training. The cost of the small validation set is the larger uncertainty in the validation accuracy. First, only blocks10+ and dense layer were trained for almost 3 epoch, next the full model parameters were made trainable. The initial learning rate of the Adam optimizer were \(5 \cdot 10^{-3}\) and it were decayed four times over the training (red dashed vertical lines). The training for 10 epochs took around six days on two Nvidia Tesla P100 GPUs.

We see jumps in training accuracy after completion of every epoch. It is so, because of the strategy to compose random batches of samples. In the beginning of the epochs, the order of images are randomized and this list will be processed by batches from the beginning to the end. The order of samples are only randomized again in the beginning of the next epoch. It means, that strictly after the first epoch, network only sees the images, that it has seen before. This is the reason of the jump in the training accuracy after every epoch.

Also, sudden decrease in the accuracy is visible after switching to the fully trainable model. The reasons here are bit of unknown, but the estimate is that probably some parameters, which were unattainable before, will be messed up by too large learning rate. It could be avoided by using lower learning rate, however it also means, that the learning rate must be again increased afterwards (otherwise the convergence would be too slow). In the end, this decrease in the accuracy is not a problem, because it recovers after approximately one epoch and it was ignored.

In the case of no-augmentation, a clear over-fitting is visible quite soon, after the second epoch. The training accuracy climbed up to 0.77, however validation accuracy stayed near 0.70. To reduce over-fitting, data augmentation and dropout were used. From Fig. 3 it is clearly visible, that it reduces over-fitting a lot, however also the validation accuracy seems to be degraded. This will be further discussed in the next section.

B. The results of the competition

The main accuracy results for the different configurations (with an without augmentation) of the model and for the different prediction methods (mean, product, ...) are displayed in the Table I. For each configuration, two different predictions were made: without test-time-augmentation (No-TTA) and with (TTA). The validation score (VAL) were calculated on entire validation set (10\% of the training data) and the leader-board score (LB) is extracted by submitting the prediction to the Kaggle competition.

<table>
<thead>
<tr>
<th>Without augmentation</th>
<th>With augmentation</th>
</tr>
</thead>
<tbody>
<tr>
<td>No-TTA</td>
<td>TTA</td>
</tr>
<tr>
<td>VAL</td>
<td>LB</td>
</tr>
<tr>
<td>mean</td>
<td>0.708</td>
</tr>
<tr>
<td>product</td>
<td>0.712</td>
</tr>
<tr>
<td>RMS</td>
<td>0.707</td>
</tr>
<tr>
<td>median</td>
<td>0.707</td>
</tr>
<tr>
<td>(x^{0.1})</td>
<td><strong>0.712</strong></td>
</tr>
<tr>
<td>max</td>
<td>0.706</td>
</tr>
<tr>
<td>firstImage</td>
<td>0.691</td>
</tr>
</tbody>
</table>

In the simplest case, without augmentation, without TTA and using only one image of the product, the validation accuracy was 0.691. It can be thought as a baseline result for the model we would like to improve. Indeed, by using multiple images and combining the activations by product or power rule (the best ones), we could improve the accuracy by two percent up to 0.712. Such model resulted in the LB score 0.7243. Using TTA however did not
increase the accuracy, on the contrary, the accuracy decreased. The reason probably is the fact, that the model was trained without augmentation: the model was not use to see horizontally flipped and cropped images.

Second configuration of the model used augmentation for training. Without TTA the validation accuracy was up to 0.710, which is basically the same as in the case of not using augmentation. However, in this case, using TTA worked as expected. The validation score increased up to 0.715.

In the competition, the model without augmentation, no-TTA and with $x^{0.1}$ scored the best (accuracy 0.72582) and guaranteed the 64th position out of 627 teams.

IV. CONCLUSIONS

In this paper Xception model was trained for image classification task. Special attention were devoted to study the effect of the augmentation (both train and test time) and to the algebraic algorithms to combine the predictions of multiple images of the same product. It was found, that TTA is only beneficial if also the model is trained with augmentation. In the case of the models, which was trained without augmentation, the TTA only decreased the accuracy. Combining the activations of the multiple images proved to be really important. From the simple algebraic expressions studied, the best performing were the power law ($x^{0.1}$) and the simple product law. Finally, in the competition, the best model was trained without augmentation, without TTA and predictions combined with $x^{0.1}$ law. The final accuracy of the classification were 0.72582 (64th of 627).

ACKNOWLEDGMENTS

Author would like to thank supervisors Tambet Matiisen, Ardi Tampuu and Raul Vicente.

Appendix A: How to do better?

After the competition many competitors added short descriptions of their solutions to the discussion forum. Here, the summary of the TOP1 solution will be presented [13]. The main strategies to achieve the best results were the following:

• Increase feature space (+3.0%): Usually image classification models are trained and tuned for ImageNet competition which has 1000 classes. Here, however the classification task is much bigger. The number of classification categories is over five times bigger. It was the reasoning behind increasing the feature space of the model (number of filters in the convolutional layers). It turned out to work really well, giving approximately additional 3.0% in final accuracy. In this work, this was not attempted, as the general understanding was, that the all the models are already configured in the most optimal way.

• Secondary dense layer (+0.5%): Usually only one dense layer is used after convolutional layers and before softmax. Here, however, two dense layers were used. The reasoning is the same, as in the previous point: the number of categories is so big, that additional features/axes of freedom are required.

• Make full use of multi-images of a product (+2%): The fact, that each product had variable number of images (1-4) made this classification nonstandard. It turned out to be really important to properly combine the predictions of multiple images for a single product category prediction. The TOP1 solution used actually four different models. One, which took only one image as an input, another, which took two images as an input and so on. All those four models were fine-tuned by training. Such strategy to fine-tune four different models resulted in additional 2.0% in the final accuracy.

• Ensemble different models (+1.0%): Using ensembles are a quite standard method to increase the accuracy. Here the predictions form ResNet50, Inception-ResNet-V2, DenseNet161, DenseNet169 and dpn92 were combined.

• Optical character recognition (+0.4%): An interesting idea proposed by TOP1 solution was to use optical character recognition (OCR) for classification of books and music (the most hardest categories). The idea was, that the title of the book is definitely extra information for classifying books and music by styles. However, the problem here was the fact, that the quality of the images were quite poor for OCR and the titles were in French (no good dataset for French). So, no actual OCR were possible. However, giving the smaller image containing the title area as an extra input to neural network improved the results by 0.4%.

• Have a lot of GPUs: This competition required a lot of resources. For a good results, the training of the full model was required which easily took a week on four high end GPUs.