EE628 Final Project: Dogs Cats Image Classification

Haixu Song
ECE Department
Major in Applied Artificial Intelligence
10446032
hsong13@stevens.edu

Wei Chen
ECE Department
Major in Applied Artificial Intelligence
10455425
wchen39@stevens.edu

***Abstract*—We used transfer learning with VGG16 to do the classification. Then we tried different epoch sizes, learning rates and optimizers to optimize the model in order to get the best result. Finally we tried fine-tuning and test time augmentation to see their performance.**

***Index Terms*—Deep Learning, Classification, Transfer Learning, VGG16, Test Time Augmentation**

I. Introduction

Web services are often protected with a challenge that's supposed to be easy for people to solve, but difficult for computers. Such a challenge is often called a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) or HIP (Human Interactive Proof). HIPs are used for many purposes, such as to reduce email and blog spam and prevent brute-force attacks on web site passwords. Asirra (Animal Species Image Recognition for Restricting Access) is a HIP that works by asking users to identify photographs of cats and dogs. It's a well-known dataset for deep learning beginners to learn and test how deep learning models work.

Fig. 1: The objective of the project

Our final project of EE628(Introduction to Deep Learning for Engineering) is subjected to let us use what we learned in this semester to do this classification task. We decided to use transfer learning to do this task since we don't have enough computing resource or time to do the training for hundreds of epochs. We made our decisions to use VGG16 with "ImageNet" pre-trained weights [1]. ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently we have an average of over five hundred images per node. With the weight pre-trained, we can do the task only by adding our own heading models then train it. This process will be far more faster comparing to training a model whose weights are random initialized.

Fig. 2: One sample from training set

In this paper, we firstly trained the model's head, which we defined on our owns. Then we tried several ways to optimize the model to get a better result. Firstly, we tried different learning rates and optmizers. Then, we tried different epoch sizes, made a comparison between epoch_size=16 and epoch_size=32. Finally, we tried fine-tuning the VGG16 model with a very low learning rate and also did test time augmentation to get the final result.

II. Data Processing

We finally decided to use Google Colab because we don't have a fast CPU nor a GPU which can run tensorflow on it. Colab is providing free GPU training which makes our work more convenient. However, after several weeks using, we find some cons of it. Firstly, there's 12 hours session limit for all notebooks. A session is actually a virtual machine on cloud. Whenever your notebook opens, the count down begins. This feature is reasonable since this is just a free app. But this means that you can't have a training process which is longer than 12 hours. There's two ways to get rid of it. The first one is simply pay google for a better survice(24 hours

limit). The second one is that you can train 12 hours, store the model weights, load weights and continue training, and so on. Secondly, whenever the session is retrieved, we have to re-unzip the data from google drive to the virtual machine. All in all, Colab is a good app to train the model and finishing our project.

According to experience, we did these to the dataset. Firstly, we divide the data into 20% validation set and 80% training set. This ratio is commonly used in machine learning. We used static random_state number because when we do further investigation like using TTA, we want the validation set the same with previous. We can do this by using the same random_state number. Secondly, considering the robustness of the model, we use data augmentation by Train_generator to make little tuning on training set [2]. All images will be changed a little bit according to the following parameters just before it was fed to the training model.

- Image rotation within range 15;
- Image shearing within range 0.2;
- Image zooming in/out within range 0.2;
- Image shifting h/v within range 0.1;
- Image random horizontal flipping;

These data are determined based on experience. Larger parameters will make the picture severely deformed and unrecognizable, while smaller parameters will reduce the robustness of the model.

Fig. 3: Data Augmentation Sample

Let's have a look at the distribution of class "dog" and class "cat". We can see from the chart that they are equally distributed.

III. Modeling

We have tried building Convolutional Neural Network from scratch. However, after 12 hours fitting, the model still not

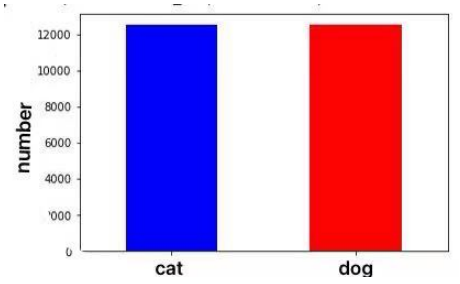


Fig. 4

showing any sign of convergence. Finally, we decided doing transfer learning as a solution.

Transfer learning is a machine learning method where a model developed for a task is reused as the starting point for a model on a second task. It is a popular approach in deep learning where pre-trained models are used as the starting point on computer vision and natural language processing tasks given the vast compute and time resources required to develop neural network models on these problems and from the huge jumps in skill that they provide on related problems. In this paper, we choose one of the most popular model VGG16 as our transfer learning model which lead to speed up training and improve the performance of original model.

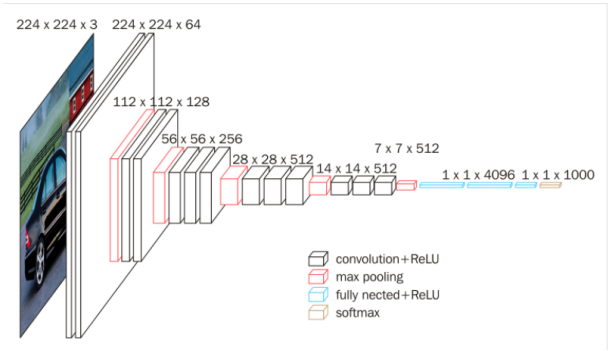


Fig. 5: VGG16 Model

As mentioned before, we use VGG16 with custom head on top of it. VGG16 is a convolutional neural network model proposed by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition". The model achieves 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. There are several distributions of VGG like VGG11, VGG13, VGG16 and VGG16. Why we finally decided to use VGG16? Because comparing to VGG11 and VGG13, VGG16 has an outstanding better performance. Meanwhile, VGG19 has too much parameters while has a similar performance as VGG16.

We customized 5 more layers on top of the VGG16. The first one is flatten layer, which can reduce the dimension of previous output. Then followed with a dense layer which has 512 neurons. After that, add a batch normalization layer and a dropout layer to accelerate the training process. Finally add

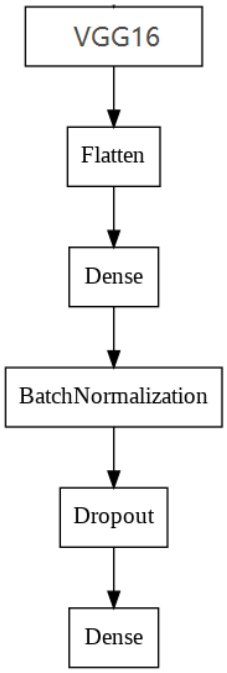


Fig. 6: Customized VGG16 Head

a dense layer which has only 2 output nodes with softmax activation function.

IV. First-time Training

We trained 50 epochs on this model with a batch size of 16. While training, we used validation generator which may scale the validation set into 0 to 1. Then we used early stop callback function which monitored validation loss with patience of 5 to prevent over-fitting. We set epoch number 50 because we tried that each epoch may take about 6 minutes to train on average. So training such a model with 50 epochs may cost 300 minutes (6 hours). Considering the later fine tuning process, we think that 6 hours is the largest epoch size we can fit. Due to the 50 epoch size, we set early stop patience 5 even though 10 to 20 is mostly used by other models since 10 too large for a training of 50 epochs. We set batch size 16 because batch size is usually exponent of 2 due to different GPU hardware resource. We will try 32 later while optimizing the model to see what's gonna be changed.

We firstly trained only the top 5 layers we added on top of the VGG16. The rest of the model (original VGG16) uses the ImageNet pre-trained weights. Every epoch costs about 6 minutes in Colab with GPU accelerated. According to Chart "First Training Result", we found that the training loss curve(blue curve above) has been declining along with the fitting process and reached the lowest point at about the 20th epoch. The validation loss curve(red curve above) fluctuates at first and then gradually decreases after 6 epochs with a lowest value 0.1422.This means that this model fitting is almost done. And the training accuracy curve(blue curve below) rises steadily during the training. The validation accuracy curve(red curve below) has a fluctuation because of the tiny validation data set. We found that the model still has room for improvement.

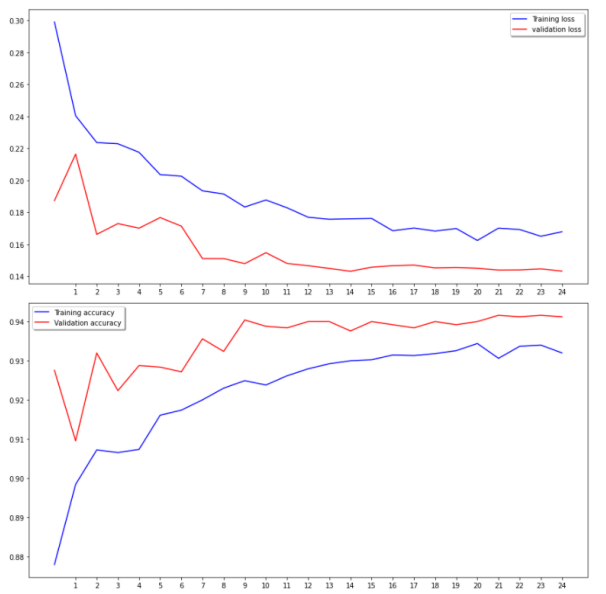


Fig. 7: First Training Result

V. Model Optimizing

There are several ways to optimize the model. Just like the idea of EfficientNet, we can change the model from 3 aspects, resolution, depth and width. We are really into this idea but due to the GPU we used and the remaining date before dead line, it's impossible to resize the model and retrain it from scratch. We can also optimize the model by selecting a better optimizer or tuning other hyper-parameters like batch size or dropout rate. Doing these needs us to retrain the model. Colab restricted us to use at most 2 sessions with GPU accelerated a day. So here's what we did after trying our best.

A. Batch Size

We investigated how batch size influenced our training and result. We tried batch size with 16 and 32. Here's the result of how different batch size influenced the performance of each model.

TABLE I: Batch Size Result

Model	Batch Size 16		Batch Size 32	
	Acc	Loss	Acc	Loss
opt. with adam	93.91%	0.1518	93.55%	0.1570
opt. with nadam	93.79%	0.1577	93.95%	0.1519
customized lr(monitering val acc)	94.15%	0.1508	94.11%	0.1432
customized lr(monitering val loss)	95.15%	0.1483	94.79%	0.1263

From the Table Batch Size Result, we see that same model has similar result when different batch sizes are given. We know that a too small batch size may result in none-convergence problem while a too huge batch size may result in too long training time. While training, we found that these model have a similar training time per epoch even with

different batch size. So our conclusion is both 16 and 32 are good enough for the model.

B. Optimizer

Optimizer is the one of the most important things to tune in a model. Different optimizer may cause different speed to converge. To some big and complicated training set, a good optimizer also may result in a slightly better result. However, our project is just trained within limited amount of time and limited amount of data set, all optimizers we tried can converge to a similar result in limited amount of time. As mentioned above, we used Adam optimizer [4], which is one of the most popular optimizers in the world. Adam was presented by Diederik Kingma from OpenAI and Jimmy Ba from the University of Toronto in their 2015 ICLR paper (poster) titled "Adam: A Method for Stochastic Optimization". This algorithm took advantage of RMSProp and AdaGrad, which means it uses an adaptive learning rate base on recent steps' gradient, and meanwhile with momentum in it. We also used Nadam optimizer, which is a different version of Adam. The only difference between Adam and Nadam is that Nadam has different momentum algorithm called Nesterov momentum. We also tried using learning decaying method with a patience of 2 to manually reduce Adam's learning rate. This is the table of our results.

TABLE II: Optimizer epoch

Optimizer	Batch Size 16	Batch Size 32
adam	18	23
nadam	21	17
customized lr(monitering val acc)	35	34
customized lr(monitering val loss)	22	30

From the table we can see that the training speed of Adam and Nadam are similar. For the Adam where we manually reduce the learning rate, the training is obviously a little slower. We analyze the reasons as follows: Adam and Nadam dynamically adjust the learning rate according to the given/default learning rate, combined with gradients of nearby steps. For our current model, this dynamic adjustment is effective enough. Even though we artificially reduce of the learning rate is feasible in some cases, but our patience parameter is too small, which may reduce learning rate when a small learning rate is not needed. This may finally result in a decrease in learning speed. We can also see from the third and fourth rows that monitoring validation accuracy is worse than monitoring validation loss. This is obviously because validation loss can reflect the quality of the model more accurately than accuracy.

Our conclusion is Adam and Nadam have similar converging speed. Manually reducing learning rate is not needed for our model.

C. Fine-Tuning

We used transfer learning in our project. The network structure and parameters of the first 19 layers of our model are all pre-trained by ImageNet. Although this helps us reduce

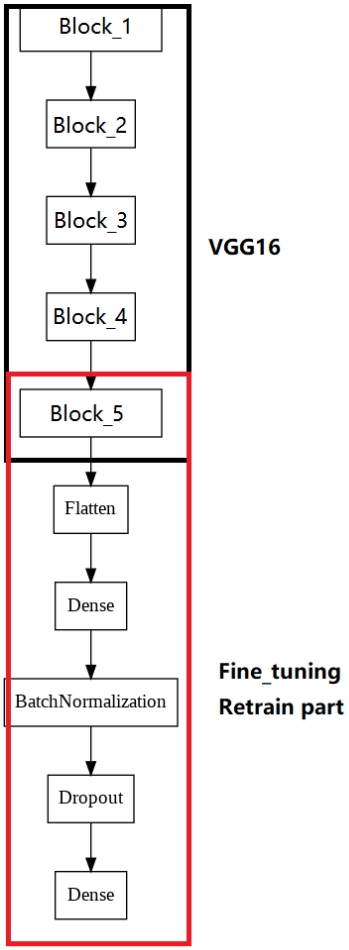


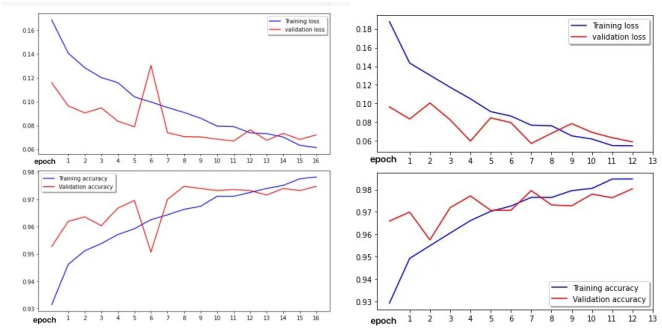
Fig. 8: Fine-Tuning

a lot of training time, the 19-layer network is not necessarily 100% suitable for the tasks in our project. At this point, we need to do fine-tuning to make this model perform better. There are several fine-tuning methods [3]. We did this: keep the parameters of the first 15 layers unchanged, and retrain the networks behind the 15 to 19 layers with a small learning rate. In other words, this time we train the last three convolutional layers in VGG16 together with the head we customized before. The training results of different hyper-parameters models are as follows:

TABLE III: Fine-Tuning Result

Model	None Fine-Tuning		Fine-Tuning	
	Acc	Loss	Acc	Loss
FT opt adam (Batch Size 16)	93.79%	0.1576	97.68%	0.0675
FT opt SGD (Batch Size 16)	95.35%	0.1092	97.11%	0.0854
FT opt adam (Batch Size 32)	93.91%	0.1518	98.03%	0.0853
FT opt SGD (Batch Size 32)	93.55%	0.1570	97.47%	0.0721

As we can see from the table and figure, fine-tuning obviously improves the accuracy of the final prediction of the model and significantly reduces the loss of the verification set. It is worth noting that we see that even if we use an old optimizer such as SGD, the effect is also very good, that's



(a) Fine-Tuning Optimizer SGD

(b) Fine-Tuning Optimizer Adam

Fig. 9: Fine-Tuning Loss and Accuracy

because we gave SGD a very low learning rate. But we can still see from the sixth epoch that even with such a low learning rate, SGD will still cause greater fluctuations. Comparing to SGD, Adam's fitting process is more smooth.

Our conclusion is that, our model gets a better accuracy through fine-tuning with a low learning rate.

D. Test Time Augmentation

Similar to what Data Augmentation is doing to the training set, the purpose of Test Time Augmentation [5] is to perform random modifications to the test images. Thus, instead of showing the regular, "clean" images, only once to the trained model, we will show it the augmented images several times. We will then average the predictions of each corresponding image and take that as our final guess. In our project, we made a validation set generator which used the same parameters with the training generator. We tested 10 times of each picture and got an average accuracy as followed.

TABLE IV: Test Time Augmentation Result

Model	None TTA		TTA	
	Acc	Loss	Acc	Loss
FT opt adam	97.67%	0.0675	97.15%	0.0846

We see that the accuracy actually dropped after using TTA technique. We analyzed our result as follwed: we used TTA in a wrong way. Often, a single simple test time augmentation is performed, such as a shift, crop, or image flip. In a 2015 paper that achieved then state-of-the-art results on the ILSVRC dataset titled "Very Deep Convolutional Networks for Large-Scale Image Recognition," the authors just used horizontal flip test time augmentation. However we did rotation, shearing, zooming, shifting and horizontal flipping. This didn't help for a better prediction.

Our conclusion for this part is: we used the TTA technique in a wrong way, and we got a worse result.

VI. Conclusion

We used transfer learning of VGG16 with a customized head to do the classification task. Then we tried different epoch sizes, learning rates and optimizers to optimize the model in order to get the best result. Finally we tried fine-tuning and test time augmentation to see their performance. The best performance we finally got is 98.03% in accuracy and 0.0853 loss on validation set.

VII. Future Work

- Based on existing model, we can try other transfer net like AlexNet, ResNet and Inception.;
- If time is sufficient, we can use EfficientNet's method to adjust hyper-parameters such as resolution, width, height, etc.;
- Adjust TTA's strategy. Reduce TTA feature numbers and only do TTA on those pictures with lower predicted probability may give us better results;

References

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in International Conference on Learning Representations, May 2015.

[2] Mi, Qing, et al. "The Effectiveness of Data Augmentation in Code Readability Classification." Information and Software Technology, vol. 129, Jan. 2021, p. 106378. DOI.org (Crossref), doi:10.1016/j.infsof.2020.106378.

[3] F. Radenović, G. Tolias and O. Chum, "Fine-Tuning CNN Image Retrieval with No Human Annotation," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 41, no. 7, pp. 1655-1668, 1 July 2019, doi: 10.1109/TPAMI.2018.2846566.

[4] Poplavskii, R. P. "On an Improvement of the Convergence of the Fourier Method." Journal of Applied Mathematics and Mechanics, vol. 24, no. 2, Jan. 1960, pp. 562–66. DOI.org (Crossref), doi:10.1016/0021-8928(60)90063-0.

[5] Sharma, Neeraj, et al. "Automated Medical Image Segmentation Techniques." Journal of Medical Physics, vol. 35, no. 1, 2010, p. 3. DOI.org (Crossref), doi:10.4103/0971-6203.58777.