CSC 370 Computer Vision Portfolio

Yuhan Wang
Prof. Nicholas Howe

Exercise #2

Goal:

For the 2nd exercise, I plan to continue on the CNN model from the 1st exercise. There are 3 parts I want to work on: 1) revise the validation & test part to confirm that the prediction is in sequence and the labels are correctly corresponded; 2) figure out ways to train more epochs and observe how the basic CNN performs; 3) implement another pretrained, more nuanced CNN architecture and evaluate its performance.

Basic CNN

Building on the preliminary model and its code, I looked into the *ImageDataGenerator* & *.flow_from_dataframe* functions because the testing score was around 0.1. No matter the model's performance, the results still yielded at a random rate. I suspected either the class id was wrong or the sequence of the test dataset was wrong.

It turned out that the *image_generator* for testing dataset can not contain shuffle or random, and I've also eliminated putting a placeholder class label for them to clear out any potential interference. Through testing with the validation dataset, it worked with the right accuracy as shown in the training process.Thus prediction on testing dataset was correct, and we had a base accuracy of 0.874 from the Kaggle submission.

Another finding from this practice is that Colab will unforeseeably get runtime disconnected after several hours of inactivity, so I can't really train the model with more than 5 epochs that lasts beyond 5 hours. Then I transferred everything to the local machine, the training process accelerated significantly and each epoch only took around 20 minutes to train. This huge difference in speed is very perplexing.

Since the training speed is faster and running on local machines can run longer without disruption, I trained the basic model (3 convolutional layers, each followed by a max pooling layer, a flatten layer, and 2 dense layers) with 10 epochs, 30 epochs, and 100 epochs on the

whole dataset. The training of 30 epochs took around 12 hours, and the training of 100 epochs took around a day and a half. The final model result from different epochs is as follows:

Model	Epochs	Accuracy
Basic	5	0.8740
Basic	10	0.9176
Basic	30	0.9221
Basic	100	0.9144

It is surprising that a CNN model with a very simple architecture achieves such accuracy. Training the model for 10 epochs yields a good result, and 30 epochs achieves the highest accuracy, which fits a popular "rule" to train a CNN with 3*classes epochs. For the 100 epochs, it is clear that the model overfits. Unfortunately, the training history was not saved properly to produce an accuracy & loss plot to visualize the training process. But through observation, it seems that 30-60 epochs yields around 92% to 93% accuracy, and after 60 the model's performance starts to become unstable and not growing.

CNN Architecture research & Inception

A 92% accuracy seemed to me a good stopping point for the basic CNN model. I decided to switch to implement a more complicated, effective CNN architecture. Last practice I tried to create an AlexNet from scratch, but it's not exactly the architecture and did not work well. Tensorflow does offer many other pre-defined architectures as well as pre-trained CNN models. I went on researching some breakthrough CNN architectures after AlexNet: VGGNet, GoogLeNet, and ResNet came out.

Model	Highlight	Pro	Con
AlexNet	5 conv + 3 ann, ReLU	fast + less computation	low accuracy
VGGNet	two 3x3 conv layers	more features from 1 strike	large computation
ResNet	shortcut identity connection	allows high epoch number	large computation
GoogLeNet	parallel multi-size conv layers	wide feature extraction	some computation

For this exercise, I aimed to quickly implement one of them and see if it achieves a better result than the basic CNN model for the next iteration of tuning. After going through the above 4 architectures, I think GoogLeNet (Inception) is the most efficient model to achieve a considerable performance for now. Inception requires comparatively the same training time as AlexNet, and significantly less computation compared to VGGNet and ResNet. Thus, I implemented tensorflow's Inception v3 module.

Since Inception v3 was built-in in tensorflow, setting up the model was not complicated. I started the training of Inception with 5 epochs, and it appeared to reach around 89% at the 3rd epoch, which is very promising. It was unfortunately interrupted, so I continued to train the Inception with 10 epochs in the next exercise.

Reference:

Singh, K. (2023, June 21). AleXNet, VGGNet, ResNet, and Inception - Karan Singh - Medium. Medium. https://medium.com/@karandeepdps/alexnet-vggnet-resnet-and-inception-11880a1ed3cd

Masood, D. (2023, October 18). Pre-trained CNN architectures designs, performance analysis and comparison. Medium. https://medium.com/@daniyalmasoodai/pre-train-cnn-architectures-designs-performance-analysis-and-comparison-802228a5ce92

Uzila, A. (2022, November 26). 5 popular CNN architectures clearly explained and visualized. Medium. https://towardsdatascience.com/5-most-well-known-cnn-architectures-visualized-af76f1f0065e

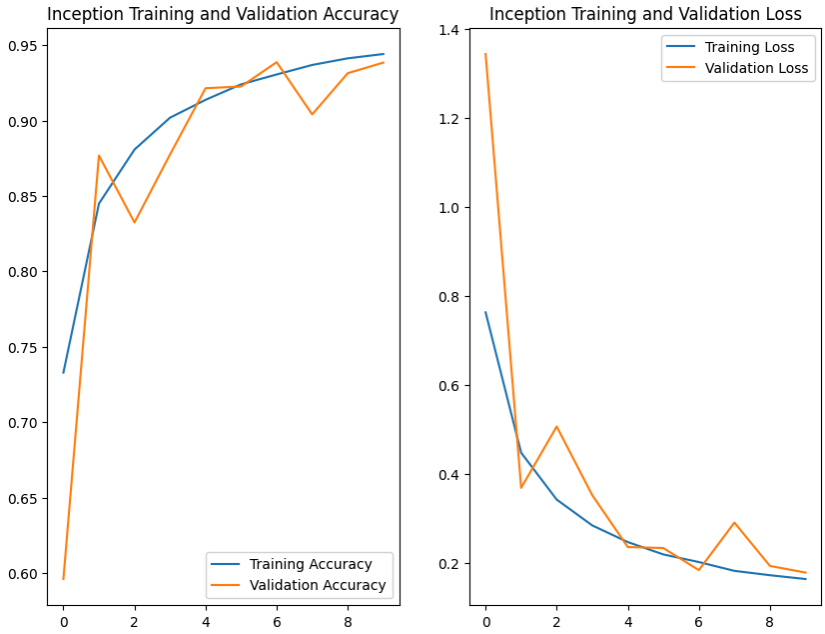
Exercise #3

Goal:

For the 3rd exercise, I plan to continue on Inception from the 2nd exercise. For the Inception model, I want to look at its performance after 10 epochs as evaluation. Beyond CNN, I also want to look at YOLO's implementation as a potential lab accompanying my presentation.

Inception & Transfer Learning

Continuing on the Inception model, this week I trained the Inception with 10 epochs. As the Inception model requires considerable computation compared to the basic CNN, the whole training process took 20+ hours. Below is the training accuracy & loss plot.



The plot shows a good trend of progress through epochs. The final prediction yields .9425 accuracy. Compared to the energy and time spent on the basic CNN model, the results balance pretty much the same, however, the inception model might have the potential to reach a better performance after more modules. Thus I will continue on this model's training with more epochs.

At the same time, I decided to look at transfer learning. Transfer Learning is to apply an already trained machine learning model in a new problem context. The idea is to exploit what has been learned in one task to the generalization in another. There are 3 approaches in transfer learning: 1) train one model and transfer learning from this model to refine it; 2) transfer learning from available pre-trained models; 3) transfer learning for best feature extractions, also

known as representation learning. In my case, the second choice is what I want to focus on. Since the previous training has required hours, even days for one result, leveraging a pre-trained model to generalize my problem seems to be the best choice.

As the Kaggle competition I participated in was in tem format, the other members were particularly focusing on transfer learning. I consulted with their experience and recommendations, and looked up the available models[1] for transfer learning in tensorflow. Eventually I decided to start with VGG16. From the 2nd exercise, it seems that VGGNet extracts the most information from its input and requires the most computation and time. Since the biggest obstacle I faced in the training is limited time, this problem might benefit the most through leveraging a pre-trained VGG.

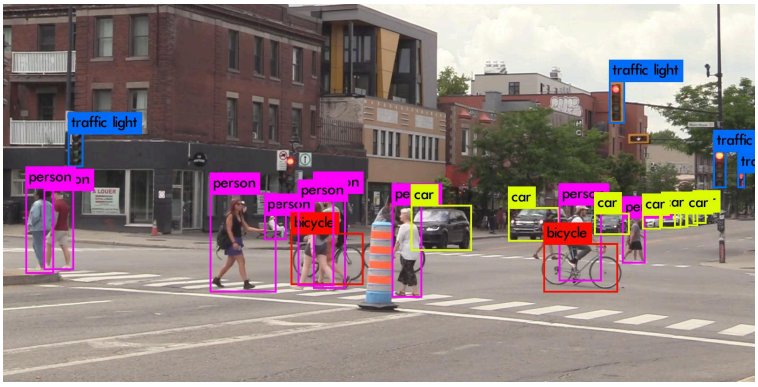
I wasn't able to perform a transfer learning training process successfully. However, through observation, it seems that the time and energy needed is considerably the same with the Inception model and the basic CNN. Since I am still unfamiliar with transfer learning and how to best tune it, I might use the next week to focus both on the Inception and transfer learning from VGG16. Since CNN training is really time-consuming, and I really hope to continue trying different models for the image classification competition this semester, I might include some CNN-relevant part as a subpart of every exercise.

YOLO implementations

YOLO, you only look once, was the paper I focused on for this week's presentation. YOLO was interesting to me as it implemented a pre-trained CNN in the context of object detection, and it is really efficient in terms of locating an object and recognizing it simultaneously. I was hoping to find some simple implementations that we can achieve within minutes for others to experience the process of object detection with YOLO.

The easiest implementation I found was from the author Redmon's website. Everything needed to be done to see how YOLO works was to clone his repository, download the weights of darknet, and then run the code. A picture of bounding boxes like this would be produced within seconds.

[1] https://keras.io/api/applications/



Beyond this easiest way, I have also looked at several other code based YOLO models. These models basically built YOLO from scratch by implementing the tools we have in tensorflow or pytorch. I wasn't able to successfully implement either one of them. However, through looking at how they construct the model, my understanding of the YOLO structure and how it actually works definitely was reinforced.

Reference:

Wikipedia contributors. (2024, January 12). Transfer learning. Wikipedia. https://en.wikipedia.org/wiki/Transfer_learning

Donges, N. (2022, September 12). What is transfer learning? Exploring the popular deep learning approach. Built In. https://builtin.com/data-science/transfer-learning

Cheng, X. (2022, March 26). Image classification with transfer learning on tensorflow. Medium. https://medium.com/mlearning-ai/image-classification-with-transfer-learning-on-tensorflow-68b6bc87ef4b

Redmon, J. (n.d.). YOLO: Real-Time Object Detection. https://pjreddie.com/darknet/yolo/#google_vignette

Nishad, G. (2024, February 27). You Only Look Once(YOLO): Implementing YOLO in less than 30 lines of Python Code. Medium. https://medium.com/analytics-vidhya/you-only-look-once-yolo-implementing-yolo-in-less-than-30-lines-of-python-code-97fb9835bfd2

GfG. (2023, June 9). YOLOV3 from scratch using PyTorch. GeeksforGeeks. https://www.geeksforgeeks.org/yolov3-from-scratch-using-pytorch/