Geovanni Ochoa

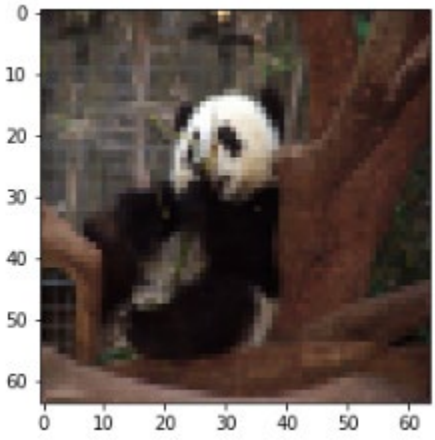
Prof. Xie CSE 144

HW 4 Report – Kaggle

Data

Given the helper functions, it was a good idea to get a feel for the data. On initiation we can see the way the color channels were split, differently than previous assignments. Therefore, it was important to split this dataset correctly so that we can work with the correct data.

Where before we had multiple rgb images we now have this to work with:



After this, we can then then split the data gotten from the train pickle file into a train and test set. We defined a preprocess function that takes in two sets, so that we can preprocess the trainX and testX sets to get the right dimensions. Similarly, for trainY and testY when one hot encoded these with to_categorical into 10 classes. This was necessary to do because we need to make sure our batches of images are up to the pre trained model's standard that we will be using.

Building

I went through many difference architectures for this HW. It was honestly really fun. I first started with a VGG16 Pre-Trained model and froze its layers. After adding my own layers and training the model on that I knew there was a huge bottleneck. I tried using ResNet50 instead and saw that it could be beneficial to freeze a majority of the layers but only leave the bottom block so I left the 143-174 layers alone (unfrozen). It was EXTREMELY important to resize the images for ResNet50's (224,224) dimensions. It took me a long time to figure this out as not doing this did not cause errors, but it wasn't making best use of Resnet50's pretrained model as that works best on images of (224x224) where ours were(64x64).

After freezing, I added this base_model to my new model and flattened it. With BatchNormalization I added 3 more Dense layers. My final Dense layer stayed at 10, but my second Dense layer changed a lot.

I ultimately ended with 1024 for dimensionality. I used relu for 2 Dense layers and softmax for the last Dense layer. This was consistent in our HW's and Exercises to be the best combos. The last step for this architecture was picking and optimizer and loss function. I ended up going with binary_crossentropy and RMS_prop based on trial and error. I switched been sparse_categorical_crossentropy, categorical_crossentropy frequently but binary proved to give the best results. I tried Adam and SGD but overall had better results with RMSprop at a learning rate of 1e-5. Very honestly most of this was just trial and error for finding the best parameters. I knew I had to start with a low learning rate so I started low and gradually moved up until I saw better results.

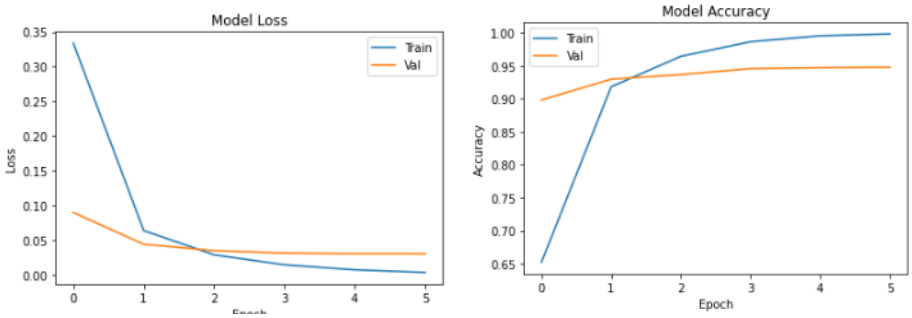
Note: I experimented with GlobalAveragePooling, Batch Normalization and Dropout which is why I left them in comments but had bad overfitting. I tried to make it work with my model but had no success. I plan to come back to this after finals to figure it out.

Training

As mentioned in colab, I tried various image augmentations and had no success. I tried brightness_range, zoom_range, width_shift, height_shift, horizontal_flip, rotation_range, etc with no success. At some point I felt like I was doing something wrong but I concluded that our data was already pretty good. Compared to situations where data comes in different sizes, formats, positions, etc. I feel like we got lucky with the images we had to work with and so image augmentation served more as a crutch. I felt like this was wrong because I did a lot of research on why image augmentation was so valuable and it did look very important. This is what I want to come back to after finals to fully understand. But after feeding this into the datagen (I left it bc I even atm am still testing new things) I fit this with a batch size of 32 and 6 epochs. 32-64 was the optimal batch sizes during hyper parameter tuning while epochs usually were around 6-25 for me. As I got better results I needed less epochs.

```
Epoch 1/6
250/250 [==============================] - 64s 180ms/step - loss: 0.5078 - accuracy: 0.4496 - val_loss: 0.0898 - val_accuracy: 0.8980
Epoch 2/6
250/250 [==============================] - 40s 158ms/step - loss: 0.0777 - accuracy: 0.9068 - val_loss: 0.0441 - val_accuracy: 0.9295
Epoch 3/6
250/250 [==============================] - 40s 162ms/step - loss: 0.0301 - accuracy: 0.9660 - val_loss: 0.0349 - val_accuracy: 0.9365
Epoch 4/6
250/250 [==============================] - 41s 164ms/step - loss: 0.0144 - accuracy: 0.9857 - val_loss: 0.0314 - val_accuracy: 0.9455
Epoch 5/6
250/250 [==============================] - 41s 164ms/step - loss: 0.0080 - accuracy: 0.9956 - val_loss: 0.0305 - val_accuracy: 0.9470
Epoch 6/6
250/250 [==============================] - 41s 163ms/step - loss: 0.0038 - accuracy: 0.9982 - val_loss: 0.0305 - val_accuracy: 0.9475
```

There was a lot of frustration during the training but getting these results made it all worth it. I plotted the graphs using the code provided for us in previous HW's. I felt this wasn't a problem as its just plotting code and isn't related to our architecture at all.



I feel like these graphs still have a very bad fit and wanted to improve them but ran out of time. After this all I just got the test data and ran it through the preprocess_input function again to make predictions with it.

Notes

Overall, this started off as a hard project but the more I researched information the more it became fun. I plan on revisiting this after finals and making better architecture. I want to be able to fully implement good data augmentation as I felt that was holding me back. I really had a fun time and would really recommend this for future students. I felt the amount I learned during this HW was super helpful and I'm really thankful to have been able to do this! I plan on doing more Kaggle competitions!