Deep Hallucination Classification Project Report

Matei Bejan, group 507

January 3, 2021

1 Machine Learning Approach

Libraries: keras, tensorflow, numpy, sklearn, pandas, matplotlib, seaborn.

1.1 Model Choice

The model we chose for completing this task was a VGG19 pretrained on ImageNet. The procedure for choosing this model was simple. We tested several pretrained networks with ImageNet weights by training them on the unprocessed data. The networks in question are as follows: VGG16, VGG19, ResNet and EfficientNet. Out of all of these we chose the one which scored the best out-of-the-box loss and accuracy, and that was VGG19.

After deciding on the base network, we then continued to customize the network in order to reduce overfit and loss. To this end, we added dropout layers after each pooling layer and batch normalization layers after every two convolutional layers of the base VGG19 model. Additionally, we set all the layers of the base VGG19 model as trainable.

We held out the last 5 layers and added two similar block consisting of the following layers, in this order: dropout, dense, leaky ReLU, batch normalization. The difference in between the two is that the former block also contained a flatten layer in between the dropout and dense layers. Moreover, the former block's dense layer contained 512 units, while the latter presents 32 units. The last layer of the model is a dense layer with 8 units and softmax activation.

The reason for which we held out the last 5 layers of the base VGG19 model was that we empirically observed that the model behaved better in this reduced form rather than in its full form.

As for optimizer, we opted for the SGD optimizer with momentum, categorical entropy loss and accuracy as metric.

1.2 Hyperparameters

Initially, hyperparameter tuning was supposed to be done automatically using AutoKeras. However, since the library is probably still in the early stages of development, it was unable to perform hyperparameter search on a network as large as our customized VGG19.

We were subsequently required to do manual hyperparameter search/tuning. Our approach was simple. We chose relatively arbitrary values for our hyperparameters then changed them one by one and only retained the values that minimize loss and maximize accuracy.

The last two dropout layers have been given a rate of 0.5. Meanwhile, all other dropouts added to the base VGG19 model have a rate of 0.25. While the 0.5 value for the last two dropout layers has been set because this is considered a good value, the dropouts within the VGG19 have been set to half of that because we did not want to leave out too much information. We tried several close values to 0.25, namely 0.15, 0.35 and 0.45, but none of them gave better results than 0.25.

Regarding the batch normalization layers, we have tried different combinations, adding them only after four convolutional layers, once after every layer and once after every two layers. We trained on all these configurations and chose the best one, which ended up being batch normalization after every 2 convolutional layers.

The Leaky ReLU layers were added later as we originally used the ReLU transfer functions that were part of the dense layers. We tried ReLU, Leaky ReLU and and PReLU and saw the results improve when using Leaky ReLU, we also tinkered with the default alpha of 0.03 parameter until we got 0.01 which seemed to be the best choice.

The number of hidden units in the last two dense layers (not counting the last layer with 8 hidden units) was more or less arbitrary, the only pattern we followed was decreasing the number of hidden units from one to the other. We tried several powers of two for both of them and stuck with 512 and 32 respectively, which seem to yield the best results.

Regarding the optimizer, we have tried both Adam, SGD and RMSProp and we stuck with the one that gave the best results, that being SGD. Out of curiosity we also tried a momentum of 0.9 and it improved results. By training with both Adam and SGD we observed that the Adam model gets a better accuracy during training, but it also greater loss on test. We will touch on the subject of learning rate in the next section.

1.3 Training

Our first learning rate choice was 1e-5 as we increased the batch size from 64 to 2048 in order to avoid overfitting and the number of epochs to 100. We then ModelCheckpoint and ReduceLROnPlateau callbacks, and thus opted for a higher starting learning rate, namely 4e-3. We obtained this value by incementally increasing the learning rate, starting from 1e-5.

Both ModelCheckpoint and ReduceLROnPlateau callbacks monitor the loss with the purpose of minimizing it.

Regarding ReduceLROnPlateau, we used a patience of 10 epochs, a factor of 0.1 and minimum learning rate of 0.5e-8, which we simply chose because it's a very small value.

The reason why we are not using early stopping is because it often ended the training prematurely, when the loss was still over 1 and the accuracy under 70%.

As an additional note, we used 800 samples for both validation and test in order to maximize to amount of data the model is exposed to, splitting data with sklearn's split method.

1.4 Data augmentation

We tried adding more data to the original 35.000 images by two methods:

- Data augmentation.

  To this end we used ImageDataGenerator from . We took the parameters from the keras documentation and we tried changing or adding other augmentation parameters.

  This approach delivered slightly worse results than the initial 70%, giving an accuracy of around around 66-68%.

- Data generation.

  We used DeepMind's Vector Quantized VAE architecture, introduced in Neural Discrete Representation Learning in order to learn latent features and generate new, slightly less accurate, replicas of the data.

  An an example for CIFAR-10 can be found here. We took this code and adapted it for our own data, changing the VAE parameters.

  We tried using this both to in pretrianing our model before exposing it to the original data and as part of the training data per se, by merging the real and generated images.

  However, this approach failed too, as the training metrics did not see an improvement after exposing the model to the generated data.

1.5 Data preprocessing

We decided to leave this section last because there is little to say on the subject.

Preprocessing consisted of dividing each image by 255 and then subtracting the mean pixel value from it in order to normalize the images.

2 Results

2.1 Customized Pretrained VGG19

Our model obtains a consistent 70-71% test accuracy and 0.89-0.9% test loss.

As it can be seen in the figure below, our model seems to be performing well for the 2, 3, 4 and 6 classes, while for the 0 and 7 classes it has poor accuracy. It also seems to confuse the 0 with 3 and 5 with 2.

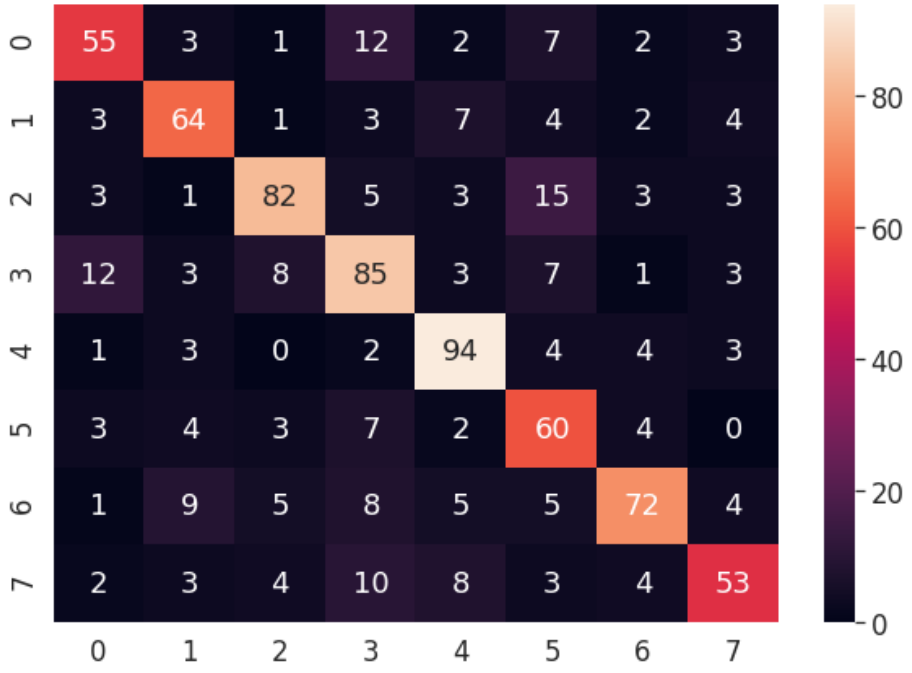

Figure 1: Customized pretrained VGG19 confusion matrix.

As it can be seen in the accuracy and validation plots on page 5, training went smoothly, the validation results following the trend set by the training results. However, the validation accuracy seems to register a small stagnation in between epochs 0 and 20, at the start of training.

2.2 Ensemble

The model ensemble consists of 9 separate customized pretrained VGG19 models (number of classes, plus one). We record the predictions of each model and then we take the most frequent prediction for each class. We only select a model if its loss is under 0.91. In other words, we train models until we 9 models with a loss lower than 0.91.

This method seems to have boosted the test accuracy on kaggle by a 0.01-0.02 percentages.

3 Figures

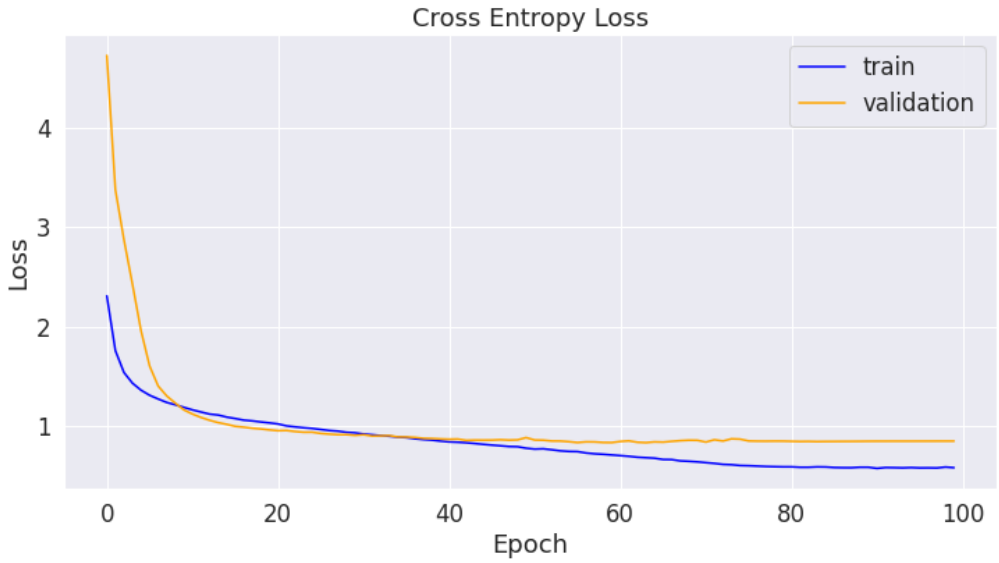

Figure 2: Customized pretrained VGG19 loss over 100 epochs.

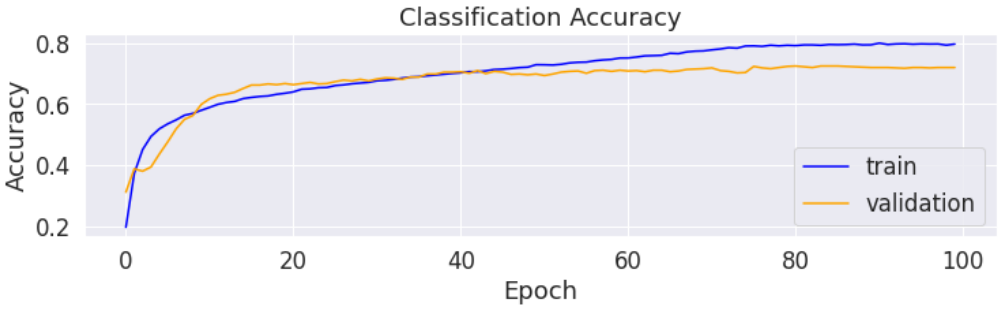

Figure 3: Customized pretrained VGG19 accuracy over 100 epochs.