Deep Hallucination Classification

Documentation

Mitrica Octavian (241)

Description

This is a deep image hallucination classification challenge in which competitors train deep classification models on a data set containing images generated by deep generative models. Competitors are scored based on the classification accuracy on a given test set. For each test image, the participants have to predict its class label.

Data

I used pandas library to convert the data into a data frame with id, label (except for test data) and path columns. The function that does that is ‘getDataframe’, which iterates through the given data files and splits them into the afore mentioned columns.

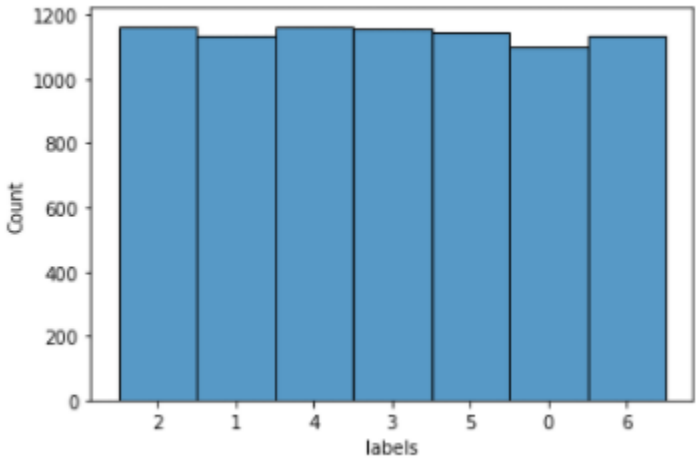
Moving forward, I was curious about how these images were looking and so I coded the ‘showImages’ function that receives a data frame and a sample size and prints some random images from the data frame we just created. Here are some samples of the images we are supposed to classify.

Train:

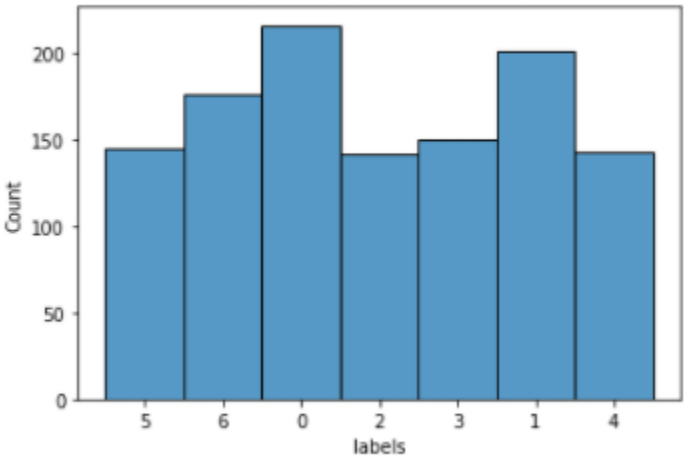
Valid:

On top of that, I also checked to see how the labels are distributed among the images using a histogram from the seaborn library.

Train:



Valid:



As we see from the charts above, we have an even distribution and I noticed there were 7 total classes which was helpful.

Dataset

I created my own ImagesDataset class inheriting the PyTorch Dataset in order to read my images and augment them. For the augmenting part I used the albumentations library. Firstly I applied some horizontal/vertical flips and a rotate to the images with a probability of 0.5. We will later try to add more and modify the probability. Secondly I read the images and I reversed the color channels with the cv2 library. After that I transposed the image and turned it into a tensor. Class finally returns the image and it's label (except for test data) as tensors.

Before going any further, we need to do a sanity check to see if the images are being preprocessed right. So I created an instance of my class and put it in a DataLoader in order to print some information in batches (we'll use batches later on to compute our loss every n images instead of doing it for every entry). I'll use the DataLoader in training later also.

```
Batch:  0
Image: torch.Size([5, 3, 16, 16])
Label: tensor([[2],
        [1],
        [1],
        [4],
        [2]])
Batch:  1
Image: torch.Size([5, 3, 16, 16])
Label: tensor([[2],
        [3],
        [4],
        [1],
        [1]])
```

We notice that the images have 3 color channels and are sized 16x16 pixels. So far, everything looks good. Now we move on to creating the model.

Model

Auxiliary functions:

Before creating the actual training model we will need some auxiliary functions to begin the training with and compute average accuracy in order to rank our model. These functions are universal and easy to modify, so we'll use them for all the other models (even if we might make small changes).

The first one is 'computeAccuracy' which is aimed to be used on train and valid datasets in order to get the accuracy of our model. It takes the model, data and batch size as parameters and returns the respective accuracy. We start by setting the model in evaluation mode and storing the data in a loader. Then we iterate through the batches and apply our model to the images resulting in the 'outputs' variable. We select the maximum probability label for each output probabilities in the 'prediction' variable. Finally, we sum the correct labels from our prediction and divide the total correct predictions by total predictions resuting in our return values which is the average accuracy of the data we just went through.

The second one is 'computePredictions'. It's similar to the one before but instead of returning the accuracy, it returns a tensor of the prediction for the last batch (which we suppose to be the most accurate one).

The third function is 'testModel'. This one is supposed to receive a model (which we want to test), the train data along with the validation data, and test the given model's average accuracy after a number of epochs (num_epochs parameter). The function also needs to receive a batch_size, the criterion parameter which is the loss function and an optimizer which is the optimizer function that updates our weights and biases in order to reduce the loss.

The fourth and final function is 'trainModel'. It is similar to 'testModel' but this one doesn't have the test data accuracy as there are no labels for them. With this function we need to save our state_dict after each epoch so we don't lose the time we just spent training the model. Then we return with a call to 'computePredictions' which results in a tensor of labels from the last batch.

Model creation:

Feed Forward Neural Network

My first approach here was to try a simple feed forward neural network as I was just starting to grasp the idea of what a neural network is. I started by defining my own classifier class which inherited the nn.Module class from PyTorch. Firstly, in the constructor we have to call the upper class' constructor as well and after that we create our first layer. The first call to linear should have our image size as the first parameter (as we've previously seen, it's [3, 16, 16]) so 3*16*16 and the next ones is a hyperparameter which we can later change. After each call to linear, we call the activation function which is in our case ReLU. The last call to linear should have the number of classes by which we want to classify the images as the second parameter, so 7 in our case. Secondly we construct the 'forward' function where we flatten our image and then we take it through the layers and return the prediction probabilities tensor.

After we did all of that, it's time to test it out. In order to do that I created an example dataset containing 500 of the train images and another one with 200 of the validation images. I ran the testing function for 5 epochs and this is the output:

```
Loading data...
Testing the model...
---Epoch: 0---
Epoch Loss: 1.9715503692626952
Training Average Accuracy:  0.122
Testing Average Accuracy:  0.15
---Epoch: 1---
Epoch Loss: 1.9616514086723327
Training Average Accuracy:  0.164
Testing Average Accuracy:  0.16
---Epoch: 2---
Epoch Loss: 1.9520771543184916
Training Average Accuracy:  0.148
Testing Average Accuracy:  0.15
---Epoch: 3---
Epoch Loss: 1.945097878575325
Training Average Accuracy:  0.216
Testing Average Accuracy:  0.21
---Epoch: 4---
Epoch Loss: 1.938171408176422
Training Average Accuracy:  0.234
Testing Average Accuracy:  0.26
```

As showing above, the accuracy ended up being too low and the loss reduction was low too after each epoch so I decided to not continue with training the model at this point. On to the next one.

Convolutional Neural Network

After that low score, it's time to move to a more complicated approach, a convolutional neural network. Again I define my own class but this time we start with a convolutional layer. I gave 3 parameters to the function Conv2d as follows: 3 – input channels (our images have 3 channels), 16 – output channels (hyperparameter) and 3 – kernel size (size of the filter). After each convolution I applied the activation function ReLU and MaxPool(2,2) which reduces the image size by a kernel size 2 and moves it by a stride of 2 and so on. To calculate the size of the resulting output image we can apply the formula:

$[(W - K + 2P)/S + 1] \times [(H - K + 2P)/S + 1]$

W – image width (in our case 16 in the beginning)

H – image height (also 16)

K – kernel size (3)

P – padding (0 - default)

S – stride (1 – default)

After the convolutional layer comes an FNN layer just like the one we had before. We need to calculate our image size after the convolutions and follow the steps from before.

Finally, in the ‘forward’ function we take the image through the network and apply the convolutions to it, flatten it like we did before and finally we apply the softmax function to the output prediction (activation function).

For testing I used the same dataset examples from before and this is the output:

```
Loading data...
Testing the model...
---Epoch: 0---
Epoch Loss: 1.9204319298267365
Training Average Accuracy:  0.283
Testing Average Accuracy:  0.25
---Epoch: 1---
Epoch Loss: 1.8624819338321685
Training Average Accuracy:  0.341
Testing Average Accuracy:  0.3
---Epoch: 2---
Epoch Loss: 1.8033864359060923
Training Average Accuracy:  0.375
Testing Average Accuracy:  0.32
---Epoch: 3---
Epoch Loss: 1.7513256534934043
Training Average Accuracy:  0.384
Testing Average Accuracy:  0.385
---Epoch: 4---
Epoch Loss: 1.7113949000835418
Training Average Accuracy:  0.409
Testing Average Accuracy:  0.36
```

As we can see, now the accuracy is going up much faster compared to the last model. In this case I decided to run the training on the whole data and this was the final output before submitting:

```
Epoch Loss: 1.3040864223241806

Final Training Accuracy: 0.508375
```

For my second submission I tweaked the preprocessing and the hyperparameters as it can be seen in the .py file and this was the output of testing:

```
Loading data...
Testing the model...
---Epoch: 0---
Epoch Loss: 1.9501590728759766
Training Average Accuracy:  0.16
Testing Average Accuracy:  0.175
---Epoch: 1---
Epoch Loss: 1.9444968700408936
Training Average Accuracy:  0.158
Testing Average Accuracy:  0.165
---Epoch: 2---
Epoch Loss: 1.9392993450164795
Training Average Accuracy:  0.202
Testing Average Accuracy:  0.165
---Epoch: 3---
Epoch Loss: 1.9318968951702118
Training Average Accuracy:  0.218
Testing Average Accuracy:  0.175
---Epoch: 4---
Epoch Loss: 1.9262062549591064
Training Average Accuracy:  0.224
Testing Average Accuracy:  0.18
---Epoch: 5---
Epoch Loss: 1.9216890037059784
Training Average Accuracy:  0.25
Testing Average Accuracy:  0.24
---Epoch: 6---
Epoch Loss: 1.913578280380794
Training Average Accuracy:  0.312
Testing Average Accuracy:  0.31
---Epoch: 7---
Epoch Loss: 1.9033888056874275
Training Average Accuracy:  0.284
Testing Average Accuracy:  0.285
---Epoch: 8---
Epoch Loss: 1.8910127480824788
Training Average Accuracy:  0.294
Testing Average Accuracy:  0.31
---Epoch: 9---
Epoch Loss: 1.8797369360923768
Training Average Accuracy:  0.314
Testing Average Accuracy:  0.3
```

It was similar or even worse than the other one but I still decided to try my luck and train this one too. After 20 epochs I submitted my prediction again and the score was just under the first one.

Transfer Learning

This was my last attempt as I started submitting predictions on the last day so I tried to make it count. I turned to transfer learning and after a lot of tries and failures I found the ResNet18 pretrained model. This one was pretty straight forward, I just created the model and modified the classification layer with my number of classes (which was 7). After running it on our testing datasets this is the output:

```
Loading data...
Testing the model...
---Epoch: 0---
Epoch Loss: 2.202502965927124
Training Average Accuracy:  0.09
Testing Average Accuracy:  0.1
---Epoch: 1---
Epoch Loss: 2.1335960626602173
Training Average Accuracy:  0.138
Testing Average Accuracy:  0.12
---Epoch: 2---
Epoch Loss: 2.0728241867489285
Training Average Accuracy:  0.158
Testing Average Accuracy:  0.185
---Epoch: 3---
Epoch Loss: 2.025028705596924
Training Average Accuracy:  0.254
Testing Average Accuracy:  0.195
---Epoch: 4---
Epoch Loss: 1.9847627798716228
Training Average Accuracy:  0.302
Testing Average Accuracy:  0.22
---Epoch: 5---
Epoch Loss: 1.9371788435512118
Training Average Accuracy:  0.324
Testing Average Accuracy:  0.3
---Epoch: 6---
Epoch Loss: 1.9018196775799705
Training Average Accuracy:  0.314
Testing Average Accuracy:  0.3
---Epoch: 7---
Epoch Loss: 1.8701676974693935
Training Average Accuracy:  0.366
Testing Average Accuracy:  0.33
---Epoch: 8---
Epoch Loss: 1.8394013952325892
Training Average Accuracy:  0.378
Testing Average Accuracy:  0.32
---Epoch: 9---
Epoch Loss: 1.8161683837572733
Training Average Accuracy:  0.406
Testing Average Accuracy:  0.36
```

It wasn't looking very good in the beginning but after I raise the epochs number, the loss started slowly coming down and the accuracy up. I decided to finally train it for 40 epochs with a batch size of 200 and this was the outcome:

```
Epoch Loss: 1.0095964059233666
Final Training Accuracy: 0.65375
```

Finally the loss and accuracy were the best I had during the competition so I decided to send it in.