MACHINE LEARNING NANODEGREE 2019

CAPSTONE PROJECT REPORT

Armando Contreras

07/27/2019

1 Definition

1.1 Overview

One of the most widely applied use cases for machine learning in real work applications is in the field of computer vision. We will focus on a specific application within computer vision that provides a solution that will classify a set of mages into a predefined set of class categories.

Image classification is a technique that is used in a wide array of applications, ranging from self-driving cars and crop monitoring. Computer vision is also used in other real-world applications such as Augmented reality or binary classifiers, but during this report we will be only going over the techniques and challenges that come with a n-class classifier.

For this project, we will use a dataset retrieved from Kaggle that contains 114 different sets of fruit images. We will apply techniques that are optimal for image classification and try to maximize our defined metrics. We will show how the same algorithm could be applied to other image datasets and showcase some techniques that improve our results

1.2 Problem Statement

We will be building a machine learning algorithm that will receive an image as input and will classify the image as a particular fruit. The image data set has been retrieved from a Kaggle dataset and the challenge is to find the best algorithm to attempt to accurately do image classification.

Convolutional neural networks are a technique that has excellent results when tackling an image classification problem. We can also test how pretrained models

perform while classifying images. As a result, we will identify which algorithm is the best suited and performs the best with our given constraints.

1.3 Metrics

The performance of our model will be determined by the logarithmic loss of the predicted class label. Our input classes are not balanced and therefore this will be a better indicator of good performance. Some of the information about why this metric will be used can be found in this article.

2 Analysis

2.1 Data Exploration

The project uses a zip file that contains 114 fruit type classifications, the raw dataset was retrieved from Kaggle. Each of those classifiers has different images of the same fruit to help train our model. Our model will be trained on each of those image classifications and label the results appropriately

Input data fields:

- Class (defined by directory name)
- Images (list of files in a directory)

Input data characteristics:

- All images are 100x100px
- Images have 3 channels for color
- Image class with highest number of images is 900
- Image class with lowest number of images is 300

The total images in the training set are 57276, and 20% will be used for validation. The test set has a total of 19548 images.

The raw dataset has a class imbalance as noted above. We will try to get a more balanced set of classes through data augmentation.

We will also check the performance of PyTorch pretrained image classification models and check if the results are acceptable for our given constraints (reference attached)

2.4 Benchmark model

We will be benchmarking our performance against VGG-16. We will compare our results with the same input and evaluate our model's performance. We will test some of the classes that exist in the ImageNet dataset: banana, apple granny smith, strawberry, orange and lemon.

From results, apple and strawberry perform very well at 100% accuracy, will banana, orange and lemon have much more poor results. We will try to improve the performance of these classes.

3 Methodology

3.1. Data processing and creating sets

The first step we will take in data processing is to load the images from both the training and test sets provided by the Kaggle dataset. There is no validation dataset so we will use 20% of the training set as our validation set.

We will use PyTorch image transform utility to do a bit of processing on our dataset. We will first scale the images to 224x224px (this is a requirement to use for pretrained models), and also apply random rotation and flip only to the training set. This will simulate having more images in the dataset and improve our training results. The model looks to have average performance when only using the pretrained model.

We will then also apply a data transform to our image sets and scale them to 100x100px. We will use this dataset to run through our own custom CNN network and also apply the same data augmentation technique as above. We will use PyTorch's random sample to split up the data between validation and training data sets.

3.2 Create class labels

The class labels for our project are defined by the number of directories in the Kaggle image set (114). We identified some class imbalance

while processing the amount of images per data class so we will have to account for this challenge when we are creating our solution. We will use the class labels to train our custom CNN as well as to feed into our transfer learning model.

3.3 Initial model architecture

For our initial custom model, we created a CNN architecture with 5 convolutional layers and 2 fully connected layers as the last steps. We will apply a max pool step of 2x2 after each of the convolutional layers as well as a relu function to remove any negative values.

We will apply a CrossEntropyLoss function while we are doing training as well as an ADAM optimizer with a 0.001 learn rate. While we are going through our training epochs we will apply these functions to tune our models ability to learn and minimize our loss after each training step.

The final architecture and hyperparameters where chosen after trial and error of different convolutional and fully connected layers.

3.4 Model training

We will run our model from scratch through 25 epochs and split our training set into batches of 20.

As the model goes through the batches of training data, we will observe the training and validation loss and only save and update our model weights when we have a better result than previous epochs.

We adjusted our learning rate to make sure to optimize our model training

3.5 Model Evaluation Validation

While doing training, we will use the 20% of the original training set to do validation on our model. We will keep track of our best validation loss and save our model when our loss decreases. The validation loss

is calculated after each epoch during the training of the model.

Here is a list of our final model that we chose:

- The shape of the convolutional filters is 3x3
- The first convolutional layer learns 16 filters, then 32, then 64, then 128 and the final layer is 256
- The pooling layers halve the resolution after each convolutional layer
- The first fully connected layer has 512 outputs, the second 114
- The learning rate is 0.0008

3.5 Testing

We will use the training set that is defined in the Kaggle “Test” directory and iterate over the batches. We will calculate how accurate the model is as well as the loss.

We ran the test dataset through our custom CNN model and we achieved 0.029 of loss and 99% accuracy. Some of the classes seem to be harder to predict since they are similar to other classes (i.e. lemon and an orange).

3.6 Model Tuning

We will try to improve our log loss as well as use transfer learning to try to improve our performance. We will adjust the learn rate that we define for our optimizer function to minimize the loss we get for our final model. We also tried different number of convolution layers as well as different kernel sizes.

We tried to improve our results using Transfer learning, but the training process took a very long time even in GPU enabled environments. After some trial and error there I was able to train a model, but validation loss was poorer than the custom CNN that was built. The code is also included in the notebook for reference, but the final results reflect the custom model that we used.

There was some trial and error on how many convolutional layers to apply as well as the learn rate.

4 Results

We trained our model for 25 epochs and the best iteration ran an epoch 17. Afterwards the training and validation loss started to increase. The best validation loss our CNN model was 0.066 and the best. After using this model to test our model was able to predict the outcome of test data with 99% accuracy.

We also included a function that can take in an image and run a predicition using the best model that we created.

4 Results - Sagemaker

During our training and tuning phase, Udacity credits ran out so I had to resort to moving my training jobs to Sagemaker. I was able to apply prior learning of the platform to improve and tune our custom CNN.

5 Conclusion

Our custom CNN architecture performs very well for a large portion of the classes that we trained. We maximized our validation loss and saved an instance of that model and got very good results.

There are some cases where our classification model has a harder time correctly identifying classes that have similarities.

Ideally, when trying to do a project as this it is essential to have a lot of training data, as well as make sure that the classes are as balanced as possible to try to achieve the best results.

It seems also that pretrained models work better when the input images are closer to the 224x224px that ImageNet requires for images to be transformed.

References

https://pytorch.org/docs/stable/torchvision/models.html

https://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html

http://www.image-net.org

https://pytorch.org/docs/stable/torchvision/datasets.html

https://www.kaggle.com/moltean/fruits

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

https://www.researchgate.net/publication/321475443_Fruit_recognition_from_images_using_deep_learning

http://wiki.fast.ai/index.php/Log_Loss