MAIS 202 - Assignment 4

Efraim Herstic, Sheheryar Parvaz, Thomas Jiralerspong

November 23rd, 2020

1 Implementation

1.1 Model

Since we already knew a Convolutional Neural Network (CNN) is particularly well suited for computer vision tasks, it was the first class of models we used. All our models were developed in PyTorch.

The initial model was based off VGG-19 using PyTorch's built-in models[1], which include pre-trained weights. Next, PyTorch made it easy to remove the classifier layer with our own simple linear classification layer. Unfortunately, this model was ineffective at classifying, giving validation accuracies under 20%. However, this was before our first data modification and augmentations, so it might have been because of that.

Next, instead of using the feature extractor provided by VGG-19, we realised that, since the problem is not much related to the classes VGG-19 identifies, we should probably use a different approach, opting for a custom CNN implementation. Naturally, the model performed reasonably well after the denoising and normalisation, with validation accuracies above 70%.

Finally, we realised the problem is in some ways similar to MNIST in what the model has to recognise. So, we decided to build a new model based off existing models with high accuracies on MNIST[2]. The model has some interesting properties, including batch normalisation after every convolution and dropout after every pooling. Thanks to this, our model generalised much better, yielding over 83% accuracy initially, and then our highest 96% after several rounds of denoising and augmentation.

1.2 Dataset modifications

We denoised our data by setting all pixels with a value smaller than 220 to 0. We also augmented our dataset to have 15000 instances of each class by

adding random dark pixels, rotating, and shifting pre-existing instances of the classes using various tools provided in PIL and NumPy.

2 Results

2.1 Model v1

Our initial shallow model with no batch normalization or dropout layers, trained using the non-augmented dataset, achieved a validation accuracy of 0.68 and a test accuracy of 0.73.

2.2 Model v2

Our deeper, more complex model, trained using the non-augmented dataset, achieved a validation accuracy of around 0.81, and an unknown test accuracy since we never submitted its predictions on Kaggle.

2.3 Model v2 - 10000 instances per class

Training our more complex model on an augmented dataset of 10000 instances per class yielded a validation accuracy of 0.90 and a test accuracy of 0.93.

2.4 Model v2 - 15000 instances per class

Training this same model on an augmented dataset consisting of 15000 instances per class, using Stochastic Gradient Descent as our optimizer, with a learning rate of 0.001, momentum of 0.9 and L2 regularization parameter of 0.01 yielded a validation accuracy of 0.93 and a test accuracy of 0.96, which was the best result we were able to achieve for this competition.

2.5 Hyperparameters

We tried varying the hyperparameters of our final model (learning rate, momentum, L2 regularization parameter), but any changes to these parameters only yielded slightly worse results, with test accuracies of around 0.95.

3 Challenges

One challenge we ran into was the unbalanced dataset containing many more instances of higher numbers than lower numbers. This initially caused our

model to learn to constantly guess the highest number (9) and therefore not learn anything useful.

As mentioned above, we solved this problem by augmenting our dataset to have 15000 instances of each class, by applying random transformations to the instances we already had. This also provided more data for our model to train on.

We tried augmenting our dataset even more, with 20000 instances per class, but our computers unfortunately did not have enough ram to store all these instances at once. We then tried to use Google Colab to utilize their free GPU and TPU access, but unfortunately, ran into time out and storage availability issues. We only had access to a free Google Colab account, and so had access to around 12 Gigabytes of ram, and so kept running into issues when generating the additional instances to train our model on. In addition, Google Colab erases the virtual machine the code runs in after every session is completed, and so all the files uploaded are wiped every time there was an interruption or after a certain amount of time. Therefore, it was a challenge to correctly utilize the Kaggle API and pseudo command line commands to clone our Github repository into the Google Colab session, in order to directly load the dataset and python files needed without having to manually load them each time.

We also tried to implement a CNN layer with attention to teach the model only to focus on the areas of the image containing digits, but were once again faced with memory constraints which we were unable to fix in time.

4 Conclusion

We are pretty satisfied with our final results for this Kaggle Competition, but we think we probably could have achieved an even higher test accuracy if we had been able to find a way to augment the dataset even more.

However, we all learned a lot about the importance of using methods such as dropout layers and a validation set to prevent overfitting. We also saw how important it was to have a balanced dataset, and were able to put into practice some of the techniques learned during the bootcamp to rebalance our dataset. We also witnessed the power of data augmentation techniques and saw how generating more instances for our dataset allowed us to significantly increase our test accuracy. Finally, we saw how memory constraints can affect the training of a model on a very large dataset and explored some options to get around these constraints.

5 Contribution

5.1 Thomas

I helped perfect the model's training process by splitting the training set into train and validation sets and by writing the code to display the validation loss and accuracy of the model at each epoch.

I also wrote the code to get the predictions of the model on the test set and put them into a .csv file that could be submitted.

Finally, I wrote the code to perform the dataset augmentations and get a certain number of instances of each class by applying random transformations to instances we already had.

5.2 Sheheryar

I worked on researching and working on the neural net model. I also contributed to the training/validation loop and helped with exploring different techniques for denoising, including thresholding the images at a given value.

5.3 Efraim

I helped with trying to test the model on larger datasets, eventually trying to utilize google colab's free GPU and TPU access to circumvent our personal computers' limitations.

References

[1] *torchvision.models* – PyTorch Documentation
`https://pytorch.org/docs/stable/torchvision/models.html`

[2] Chris Deotte (@cdeotte on Kaggle) *25 Million Images! [0.99757] MNIST*.
`https://www.kaggle.com/cdeotte/25-million-images-0-99757-mnist`