Fundamentals of ML Final Project: CNN Image Classifier

Richard Bailey, Augusto Cohn

April 22, 2022

Abstract

Image classification is a field of high interest within the study of deep learning. Classifying images that have lots of noise and that are very dirty is one of the hardest tasks in this field, which we must face in the class collected data. Our goal in this paper is to acquire a CNN architecture capable of classifying the hand written characters data the class collected above 90% accuracy. We demonstrate the process through which we went, both with data preprocessing and architecture creation/testing, to achieve our final transfer-learned CNN architecture that accomplishes our goal.

1 Introduction

1.1 Experimental Overview

Our experimental design first started with creating our CNN architecture and preprocessing of the image data to pass to the architecture. After many trial and errors with different types of image preprocessing techniques, including morphological operations, blurring, thresholding, removing lines, and applying boundary boxes, along with different architecture types, our highest validation accuracies were around 79%. This is clearly not satisfactory, so we decided to shift our focus away from preprocessing of the data and move into focusing on transfer learning within the keras API.

1.2 Literature Overview

[1] Morphological Transformations
The dataset contained images that had varying levels of clarity and focus. Using morphological transformations allowed for the dataset to become clearer through the use of erosion, dialation, opening and closing.
[2] Xception with your own dataset
Our model utilizes transfer learning with the pretrained Xception classifier. There is some tuning that must be done to adapt Xception to work with our dataset comprising of 10 classes and this GitHub page provided necessary code to train Xception on our dataset.
[3] Convolutional Neural Networks with TensorFlow
The Convolutional Neural Network documentation with TensorFlow is a necessary for building a neural network using the TensorFlow framework. The process and structure of our classifier was built on the TensorFlow library.
[4] Image processing to improve accuracy of CNN
Blurring and thresholding are important operations in image processing for better accuracy in a CNN. Many datasets, including ours, contains noise and unnecessary features that when removed, can result in better accuracy of your model. The practices outlined in this article were used to train many different iterations of our final model including median blurring which ended up being the only image processing practice used for our final iteration of our preprocessing routine.
[5] Layer weight regularizers
The dense layers connected to the output layer can make use of regularization to avoid overfitting to the training data. The code outlined in this article was used to implement regularization in the dense layers before the output.

2 Implementation

2.1 Data Preprocessing Methods

This section contains a list of all of our used data preprocessing techniques and what the goal of each is in the preprocessing structure.

The following are the methods we used in data preprocessing:

- show: displays a single sample.
- invert: inverts the color of an image so the character is light with dark background.
- min_max_scale: applies a min max scaling to an image.

- brighten: enhances brightness of light pixels and dims brightness of dark pixels at a set threshold.
- blur: applies a median blur to an image.
- morph_close: makes gaps in lines smaller.
- morph_open: makes gaps in lines larger.
- morph_dilate: increases the width of character lines.
- morph_erode: decreases the width of character lines.
- transform: zooms in on the pixel space by a set value to eliminate unnecessary padding.

We combine the above techniques into what we found to be the most optimal data preoprocessing sequence. In our method preprocess, we apply the methods in the following order: $transform \rightarrow invert \rightarrow blur \rightarrow erode \rightarrow dilate \rightarrow open \rightarrow close \rightarrow MinMaxScale \rightarrow brighten \rightarrow MinMaxScale \rightarrow resize \rightarrow stack$

We found this to be the most optimal way to preprocess our data to provide our CNN architecture the cleanest data without over-preprocessing it to the point where it makes the data worse. The process consists of the following: a slight zoom in to eliminate unnecessary padding, color inversion to allow morphological operations to work better, a median blur to allow for erosion to then work, followed by a common sequence of morphological operations in dilate, open, and close, then brighten to enhance bright pixels and dim non-bright pixels, resize to a more appropriate size for computational expense reasons, and ending with stacking each sample three times so the data can have a matching channel size for the transfer learning, which will be explained in detail later in this report.

After applying this sequence of preprocessing operations in our preprocess method, we augmented our data via rotations in the augment method. This is what resulted in our final training dataset to train our transfer-learned CNN architecture.

2.2 CNN Transfer Learning Architecture

This section explains how we implemented our transfer learning architecture tailored to be trained on our data.

We used the built in Xception functionality in the Keras API to load in the pre-trained weights from the ImageNet deep neural network. After loading in these weights, we removed the output layer. Then we added three dense layers of decreasing width and an output layer. We then trained our modified transfer-learning model on our preprocessed data to get our final resultant model.

In order for the transfer learning to work, we had to change our input data from having one channel (gray scale) to having three channels (RGB) so it would have compatible dimensionality. We accomplished this by applying a stack method in our data preprocessing.

We trained our transfer-learned CNN model with the entire train dataset, with a sufficient number of epochs, and then saved the weights. These saved weights are the weights of our final model that is used in our test section.

2.3 Testing Model

We have included test functionality in our posted Jupyter Notebook at the end, under a section labeled test section. If you would like to test your own test data on our trained model then one must fill in file paths correctly where prompted to and then run all cells in the section. It will evaluate our model on the imported hand-written character data.

The only parameters that need to change are the file path strings to import the data; the remaining is taken care of. After doing so, running all cells in that section will apply the appropriate preprocessing techniques and will subsequently evaluate the model on that data by loading in the weights of the model we trained.

3 Experiments

In attempting to accomplish our goal of achieving a CNN that could achieve at least a 90% test accuracy on our hand-written character data set, we first attempted to clean and preprocess our data using different image preprocessing techniques and to find a good starting CNN architecture off of which to build.

3.1 Experimental Preliminaries

To find a functioning architecture, I first created a smaller CNN architecture inspired by VGG-16 to train on MNIST data. The purpose of doing this is because MNIST data is clean, to get experience with the Keras API, and to get a starting place to create an architecture for the data for this project. The CNN architecture I found to work best was simple: $Input \rightarrow Conv2D \rightarrow MaxPool2D \rightarrow$

$Conv2D \rightarrow Conv2D \rightarrow MaxPool \rightarrow Flatter \rightarrow Dense \rightarrow Output$. This architecture yielded a 99.2% test accuracy on MNIST data and thus provided a good starting architecture to test for the data of interest.

Before the preprocessing of any data occurred, we devised a data augmentation method we knew we would apply after any other preprocessing, that of rotating the images. After any other preprocessing and before any testing on the CNN architecture, we applied a 90, 180, and 270 degree rotation to each image and added it to the data with their corresponding labels. This augmented data would allow for more robustness in our model by increasing its identification capacity of rotated images.

3.2 Architecture Training and Evaluation

In order to run and evaluate our architectures and preprocessed data, we constructed a few methods that accomplish this task.

The first method we implement is PerfEvalCustomCallback which contains multiple evaluation metrics to pass as callbacks to the fit method when training the architectures. PlotTrainingPerf displays the validation metrics graphically so we can visualize how well our model is performing. EvaluateModel is the method we create to take the data and subsequently graph the validation data and predict how well the model performs on the test data.

To train the architectures, we compile the model with the desired architecture, loss, metrics, and optimizers. We then fit the model with the appropriate training, validation data, and custom callbacks.

3.3 Initial Experiments

Our initial data preprocessing consisted of morphological operations using the CV2 library. In the pixel space we would: $Invert \rightarrow MinMaxScale \rightarrow Dilate \rightarrow Open \rightarrow Close \rightarrow Resize$. Applying this series of morphological operations to our data and then testing it on a few different CNN architectures resulted in a maximum test accuracy of 77%. This clearly did not meet expectations for test accuracy. With suspicions that the model was not deep enough, overnight we trained a model with a significantly higher number of convolutional and pooling layers, which resulted in only a small boost to 79%. This lead us to conclude that the issue existed in the data, thus requiring a greater focus on preprocessing of the data.

The subsequent few days we spent on attempting to improve the preprocessing of the data. We tried numerous techniques and permutations of these techniques in attempting to find the optimal data preprocessing sequence of methods. In addition to morphological operations, this included blurring, removing lines, thresholding, and applying boundary boxes. To be able to apply boundary boxes correctly, which had the goal of zooming all images in to fit the pixel space in a more equally distributed way to allow for better classification, we first needed a way to binarize the pixel space. Boundary boxes work by finding the pixel value closest to each edge and forming a box around that space. We first attempted to add a user defined threshold, but this would either lose too much information or not eliminate enough information when applying the boundary box. We saw improvement when we then attempted applying a Gaussian blur and then using otsu's technique to threshold and binarize the pixels. The reason why otsu worked better was due to the fact that otsu uses statistical data about each image before applying the threshold. We found that the boundary box technique worked best when applying otsu's first. Combining this technique with all other permutations of morphological operations and feeding those different permutations into different CNN architectures, we never received a test accuracy that exceeded 73%. So due to this, our model actually regressed.

Upon inspection of the data, it was due to the fact that although the preprocessing technique cleaned a lot of data, it made some bad data even worse, which we conclude hurt our model overall. In attempting to avoid hurting dirtier data, we first applied a remove lines method that provably was able to remove both horizontal and vertical lines from many images, but had the same effects as the otsu's thresholding. The remove lines function cleaned many images but made a small number of bad images worse, which hurt our model overall when combined together with morphological operations, seen in a maximum test accuracy of 69%. We attempted running the same data without morphological operations, with only removing vertical lines, and only removing horizontal lines, which all resulted in models that performed in an inferior manner. It appeared to us that attempting to preprocess the data to a larger extent actually ended up hurting our model.

3.4 Solution

From countless hours of failure in trying to extend the data preprocessing to be more robust, we suspected that we would need to have a differ-

ent approach. This is the point at which that we decided to apply transfer learning with a simpler data preprocessing sequence, that way the deeper and more robust model could handle the irregularities and noise in the data that we were unable to remove effectively.

3.5 Transfer Learning Experiments

We followed the Keras developer guide for transfer learning, along with a few online resources, to aid in the development of our transfer learning model. The data we used to train our transfer learning CNN was found by finding the data that applied best on a non-pre-trained CNN architecture that doesn't utilize any sort of Otsu thresholding or line removal since it appeared that those techniques ended up hurting our model more than they helped it. The sequence of preprocessing operations we applied is explained at the end of section 2.1.

CNN architectures are designed such that the the layers that include convolutions exist primarily for feature extraction of the images. This is why we transfer learn those layers, because we wish to keep those layers intact for feature extraction on our dataset and start at weights similar to what is desired. We add three dense layers before our output layer so that our added architecture can learn the classification on our dataset with respect to the feature extraction layers of the transfered layers. Our hidden layers do not use regularizers because we first tested our transfer-learned CNN without regularizers and then also trained it with regularizers to see which way was superior, and we found that no regularizeres performed better. The number of neurons in the hidden layers decrease in width to follow the bottleneck strategy. Our activation function in our added hidden layers is the ReLU activation function since it is the best performing activation function in CNN hidden dense layers. Our output layer's activation function is softmax because that is the best for output layer multiclass classification. We use categorical cross entropy loss for the loss function also because it is a multiclass problem. Nadam is used as the optimizer because it is proven that adaptive learning rates are extremely effective.

With this clean data preprocessing and transfer learning CNN with a few of our own added dense layers, we found our solution. The model took an extremely long time to train (around 15 minutes per epoch) but gave us by far the best results, 93% test accuracy, which was satisfactory in meeting our goal of reaching at least 90% test accuracy.

At the end of our CNN training and architecture section in our Jupyter Notebook, we see that in our performance evaluation graphs that we achieve a 93.784% test accuracy on our data. The accuracies increase and the losses decrease, but not to the point of overfitting. Thus we have found a suitable model to train the entire training dataset on. This is what we do to receive our weights and we have uploaded the training weights onto canvas, as well as made them the loaded in model in our test section in our Jupyter notebook.

4 Conclusion

Our goal of creating a CNN architecture to classify our data at 90% or higher validation accuracy was a great challenge. We have demonstrated the struggles through which we went in obtaining our model, including our implementations and experimental processes. The simultaneous goals of preprocessing the data effectively as well as creating a robust and effective architecture took many long hours to meet. Through simple data preprocessing techniques, along with applying a strategy of transfer learning using ImageNet weights, we were able to construct a CNN architecture that met our overall goal of achieving above 90% test accuracy.

5 References

[1] B. K. Kuguoglu, "How to use image preprocessing to improve the accuracy of Tesseract," Free Code Camp, 06-Jun-2018. [Online]. Available: https://www.freecodecamp.org/news/getting-started-with-tesseract-part-ii-f7f9a0899b3f/. [Accessed: 22-Apr-2022].

[2] "Convolutional Neural Network (CNN)," TensorFlow, 01-Jan-2022. [Online]. Available: https://www.tensorflow.org/tutorials/images/cnn. [Accessed: 22-Apr-2022].

[3] "Layer weight regularizers," Keras. [Online]. Available: https://keras.io/api/layers/regularizers/. [Accessed: 22-Apr-2022].

[4] "Morphological transformations¶," OpenCV, 2016. [Online]. Available: https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html. [Accessed: 22-Apr-2022].

[5] otenim, "Xception with Your Own Dataset," Github, 06-Dec-2019. [Online]. Available: https://github.com/otenim/Xception-with-Your-Own-Dataset. [Accessed: 22-Apr-2022].