Julian Sneve

Final Report:

Computer Vision Digit Recognition Using Street View Housing Numbers

Problem Statement

One of the most attractive and popular fields of artificial intelligence and machine learning today is computer vision. Computer vision entails the training of computers to interpret and understand the visual world. Using digital images and deep learning models, computers can identify, classify, and react to what they see. This technology is used in many facets of today's world from simple tasks we take for granted such as facial recognition to open our smart phones to life saving cell recognition software that aids in the hunt for cancerous cells in patients.

In this project I created fairly simple convolutional neural network (CNN) model that is able to recognize digits in images. However simple the model may be the goal of this project was to create a jumping off point for creating more robust models in the future that can be edited to recognize images from any set of images.

Data Wrangling

The data I utilized for the project is the Street View Housing Numbers dataset (SVHN) maintained by the Computer Science Department at Stanford University. This dataset is a collection of over 600,000 images of house numbers collected from Google Street View images. This dataset is popular for training models due to the numbers being irregular in many ways and set in natural sceneries. The data is available in two datasets structured in different formats. The original images with character bounding box dimensions make up the first dataset and are saved all-in-one as MATLAB MAT-files. For the second, the images have been cropped to narrow the area around the digits and saved as separate csv files for images and labels.

For my own learning purposes, I worked a bit backwards with how I utilized the data. First, I took a more direct route to model creation and used the second dataset with the cropped images to create a CNN model. Then, essentially starting from scratch with the original images to learn how to edit the images to the level of the cropped images dataset. This report will be written following the programming hierarchy from data wrangling to pre-processing to training and modeling, rather than the backwards method I took.

Data Pre-processing

After downloading the data from Stanford's SVHN website I had to familiarize myself with the file formats in which the data was saved. The data was saved as MATLAB MAT-files compressed into gzip files. This required the files to first be extracted from the gzip files and then converted to HDF5 files which are similar to dictionaries. This format would make the data much easier to work with.

Exploring the files, I find the data is split into three groups: the images, bounding box dimensions, and labels. Extracting this data separately required defining multiple functions to iterate through the files.

Once the extraction was complete, I was then able to define another function to take the separate items from each file and merge them together in proper formats to produce an image with labels and bounding boxes around each digit. While this was a great step in the right direction, I next had to combine the separate bounding boxes into one (Figure 1). This would give me the dimensions for cropping the images (Figure 2).

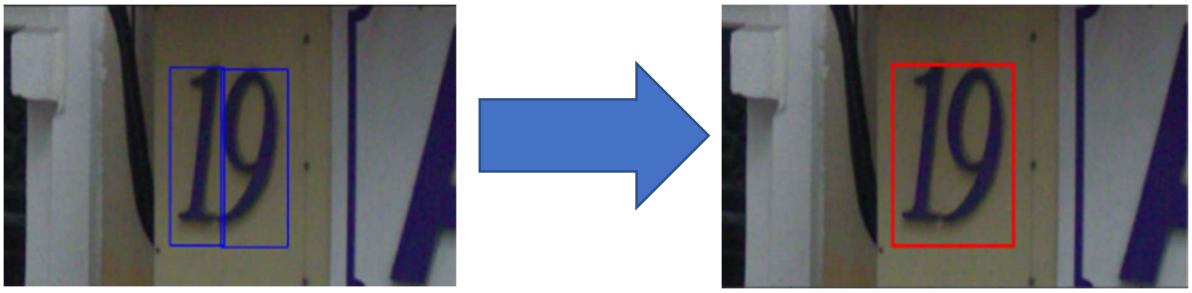

Figure 1: Multi-bounding Box to Single

Figure 2: Cropped Image

Next came the final pre-processing steps of converting the pixel data of each image to numeric data. This was done by first converting the data to numpy arrays, then creating a dataframe utilizing the file name of each image paired with the pixel values. It was quite the time consuming process. Once this was complete the dataframes were saved as csv files.

The last steps before model development were to convert all of the values to datatype float32 and binarize the labels. This would ensure the pixel data would work within the model and the labels would be able to be recognized correctly for categorical crossentropy.

Model Development

Doing research into computer vision models I decided to create a CNN using a Keras functional API. My decision was made based on the simplicity of code needed to run Keras and the flexible functionality of the functional API.

After multiple iterations of models, I finally reached the performance I was working toward with the model you see in the summary below (Figure 3). The final model consists of a core input layer, four 2D convolutional layers, four batch normalization layers, four dropout layers, three 2D max pooling layers, and the dense output layer. These layers create a total of 1,180,119 parameters, 1,179,671 of which are trainable.

```
Model: "model"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 32, 32, 1)]       0
_________________________________________________________________
conv2d (Conv2D)              (None, 32, 32, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        9248
_________________________________________________________________
batch_normalization_1 (BatchNor (None, 32, 32, 32)     128
_________________________________________________________________
max_pooling2d (MaxPooling2D)    (None, 16, 16, 32)     0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
batch_normalization_2 (BatchNor (None, 16, 16, 64)     256
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)  (None, 8, 8, 64)       0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 128)         73856
_________________________________________________________________
batch_normalization_3 (BatchNor (None, 8, 8, 128)      512
_________________________________________________________________
max_pooling2d_2 (MaxPooling2D)  (None, 4, 4, 128)      0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 512)               1049088
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 11)                5643
_________________________________________________________________
dense_2 (Dense)              (None, 11)                5643
_________________________________________________________________
dense_3 (Dense)              (None, 11)                5643
_________________________________________________________________
dense_4 (Dense)              (None, 11)                5643
_________________________________________________________________
dense_5 (Dense)              (None, 11)                5643
=================================================================
Total params: 1,180,119
Trainable params: 1,179,671
Non-trainable params: 448
_________________________________________________________________
```

Figure 3: Model Summary

After building the model I compiled the model using the following argument settings: loss set to categorical crossentropy since we're predicting multiple labels, optimizer set to Adam for computational efficiency and lower memory requirements, and the metric I wanted to report is accuracy. Also, after running the model a few times I decided to add an earlystopping callback that monitors the validation loss metric with a patience of eight.

Once the model was built and the parameters set, I fit the model to the training image and labels in batch sizes of 128. As planned, the validation images and labels were used for the validation data. I also set the number of epochs to 100, but as we'll see in the results the earlystopping callback was activated well before reaching that number of epochs.

Model Training

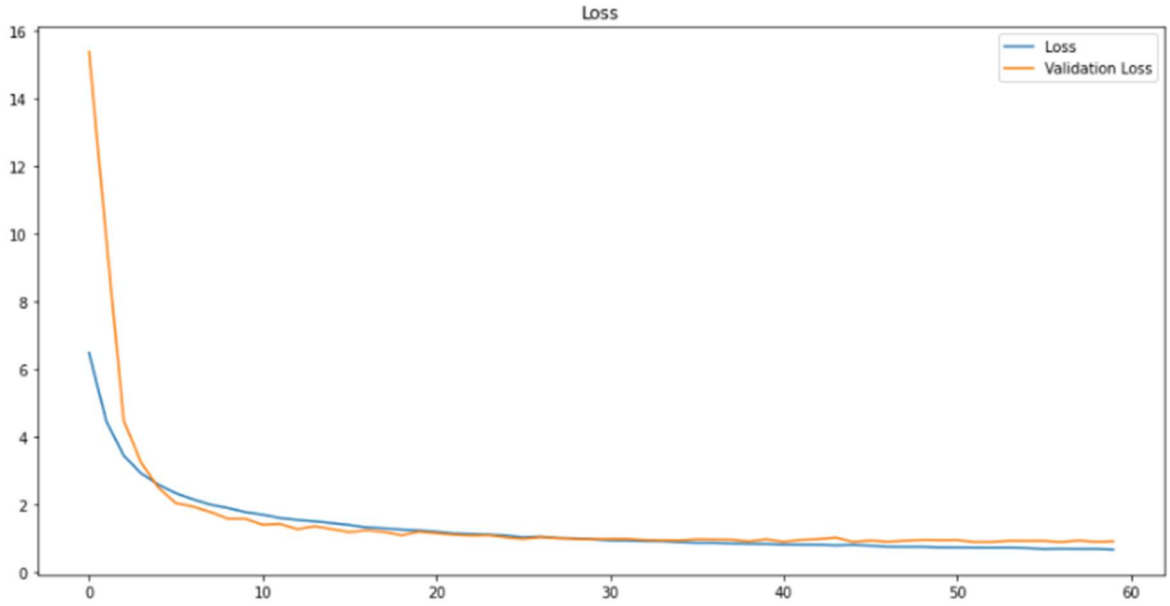

Figure 4: Loss

As you can see above (Figure 4) the model the earlystopping callback was activated and stopped the model's fitting progress at 59 epochs after the validation loss had pretty much flat-lined. This gave the model plenty of epochs to run through in order to increase both the training accuracy (Figure 5) and validation accuracy (Figure 6) to 91.51% and 91.40% respectively without overfitting the model.

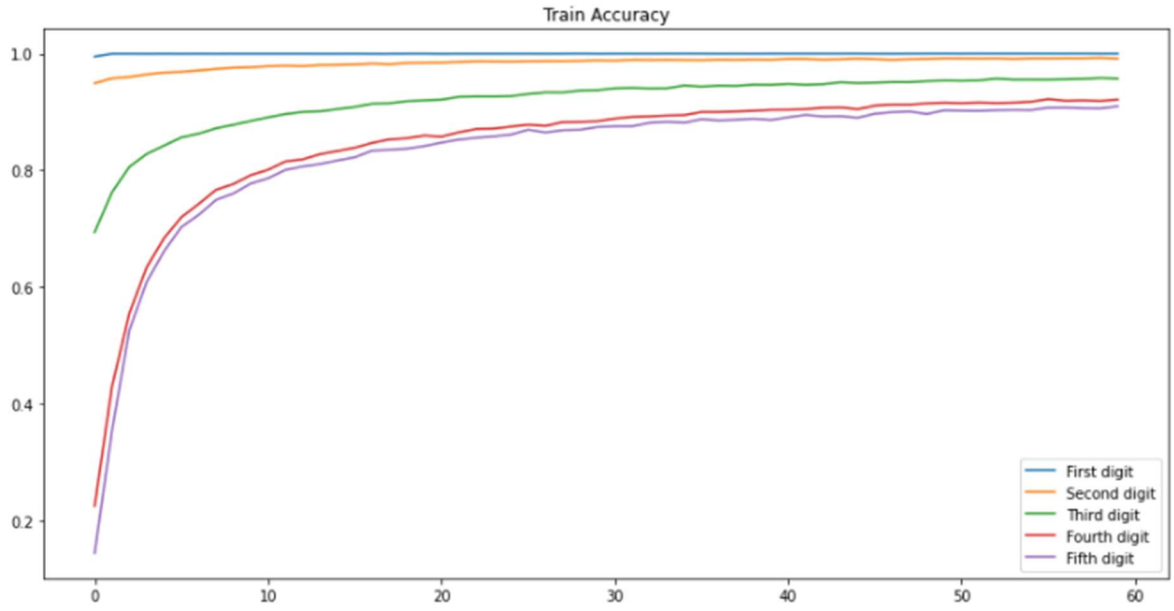

Figure 5: Training Accuracy

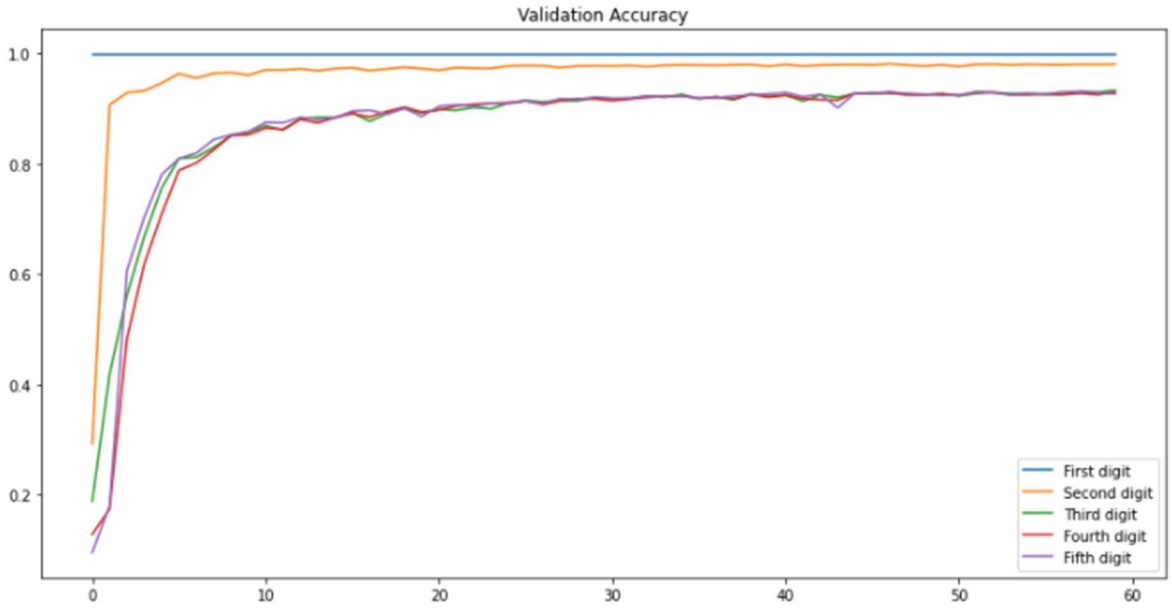

Figure 6: Validation Accuracy

Model Performance

With the model parameters set and the training and validation performance is to an appropriate level I now run the model on the test data. Evaluating the model's performance on the test data we see that for digits 1 and 2 the model predicted the digits with nearly 100% accuracy. Their accuracy came in at 99.98% and 99.58% respectively. The model's performance did drop through digits 3, 4, and 5, though only slightly with digit 3 accuracy at 95.68%, digit 4 accuracy at 91.02%, and digit 5 accuracy at 88.83%. Overall, the model was 95.02% accurate.

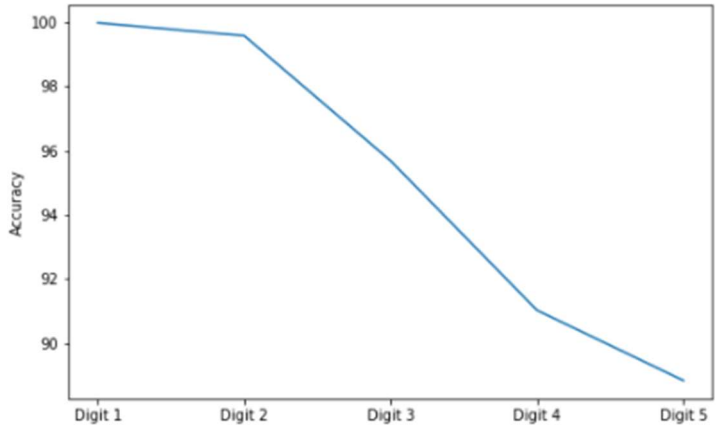

Figure 7: Test Accuracy

Future Plans

Going forward I would like to take the opportunity to grow on what I've learned with the computer vision project. This model performed on the data first being manually converted to a dataframe. My goal would be to create a model either using YOLO or something similar that doesn't

require this step and is able to make predictions based solely on the image itself. There is also plenty of opportunity to run models such as this through cloud services such as Google Colab or AWS. Training these models have definitely put my computer to the test.