Project: Counting Coins

by Jadon Manilall (815050)

I. Introduction

THE process of counting coins has been around since the dawn of capitalism. And for most of human history it has remained a to be a somewhat manual process.

Recently however, a lot more of effort has gone into automating this mundane task. Achieving this level of automation usually involves using the coins physical attributes such as it's weight and area to determine and classify it's value. While this approach benefits from a high level of accuracy, it can be costly to implement. Furthermore, these systems do not have the ability to easily to adapt to a different coin set other than what it was designed for.

In this report, we aim to implement an image based system to automate the process counting coins. The approach implemented takes place in 2 stages. The first, is the coin detector stage which takes in an image and uses an object detector (trained to detect coins) to find and segment out each coin in the image. The second stage then takes in those detected coins and attempts to classify the coin according to it's value.

The next section of this report goes on to describe the methodology of the techniques used to solve this problem at a high level. Section III then provides details about how the provided image data was structured and used in each stage of the implementation. This section also highlights some key concepts needed to train both the coin detector and the coin classifier and lists the technologies used to implement these concepts . Then the results and analysis obtained from testing the implementation is presented in Section IV. Finally we conclude by discussing some major issues that this implementation faces and how these issues are commonly solved.

II. Methodology

As stated above, the approach taken for this project consists of 2 main stages. The first stage takes in an image & then uses a **HOG+SVM** (Histogram of oriented gradients with a Support vector machine) object detector to identify & crop out all regions of an image that it "sees" as coins. Then, the second stage uses a **CNN** (Convolutional Neural Network) to take in each of those cropped images and then attempts to assign a value to it.

A. Stage 1: The Coin Detector

Initially, quite a number of methods were considered for this stage of the project. One of these approaches included using a shape detector namely, the Hough Transform tool to detect the circular shape of coins in an image. The thinking was that seeing as coins are usually circular in nature, this approach should be able to detect the circular objects(coins) while ignoring all non-circular objects (rulers, pens, student cards etc.). While this approach seemed to have some level of success in detecting the coins, it quickly showed it's naive nature when it was tested against images in the dataset that had some perspective to it. The perspective made the coins in the images look like ellipses instead of circles which in turn led to a high rate of incorrect detections. Thus, due to the nature of the task at hand, this approach was abandoned altogether in favor for a much more robust technique.

1) The HOG+SVM Object Detector:

In the paper [1] presented by Dalal et al. A new method for object detection was presented. This new method used a combination of HOG Descriptors and a Linear Support Vector Machine (SVM) to achieve nearly perfect results when tasked with identifying human pedestrians on a busy street.

Although the exact implementation of this approach presented in [1] is quite complicated, the general outline can be broken down into the following steps:

- **Step 1:** Given a set of training images containing the objects to detect(eg. coins). Take P positive samples of the object in interest and extract its HOG descriptors.
- **Step 2:** Take N negative samples from a negative training set which does not contain any objects of interest.
- **Step 3:** Train a linear support vector machine on the P positive and N negative samples.
- **Step 4:** Then For each image and each possible scale of each image in the negative training set, apply the sliding window technique and compute HOG descriptors and then apply the classifier. If the classifier incorrectly classifies a given window as an object (ie. a false-positive), record the feature vector associated with the false-positive patch along with the probability of the classification. This approach is called hard-negative mining.
- **Step 5:** Take the false-positive samples found during the hard-negative mining stage, sort them by their confidence (i.e. probability) and re-train the classifier using these hard-negative samples.
- **Step 6:** Obtain a trained object detector.

B. Stage 2: The Coin Classifier

For this stage of the project a convolutional neural network was chosen to classify images of coins. The Reason for this choice stemmed from the high amount of variability that could exist even in a single class of the image data(seen in Fig 1), as any two coins with the same value could look vastly different in input image depending on where it's placed, lighting effects etc. thus manually designing the right features to use for the classifier would be an extremely difficult task.

1) The CNN Classifier:

Pioneered by Yann LeCun, Convolutional neural networks

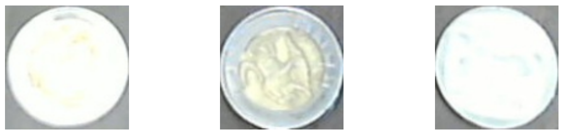

Fig. 1: Three separate coins from the same 5R class

have easily been one of the sharpest tools in the computer vision toolbox.

One of the major advantages that comes from using these networks is that they handle the feature extraction for you. Meaning that, provided you have enough of training data, these networks can learn to find the features that identify a particular class, automatically.

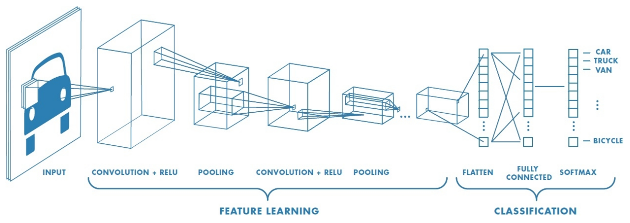

Fig. 2: General structure of a CNN

Although the exact details of these networks may vary depending on the problem at hand, most, if not all of them, can be broken up into a Feature learning phase and a Classification phase, seen in Fig 2 .[1]

The Feature Learning phase can be further broken up into a series of ”Convolutional Layers” which itself is made up of the 3 fundamental operations.

1) **Convolution:** The convolutional operator uses **k** filters, or kernels, of size **n x m** (32 and 3x3 in our case) to compute the dot product between the filter and a region of the image using a sliding window. The output from each these ”Convolutions” form the entries in the activation matrix.
2) **Activation (ReLU)** ReLU or Rectified Linear Unit is a simple, non-linear operation that is applied per pixel and replaces all negative pixel values in the activation matrix by zero. By doing this, we introduce non-linearity into our model, which arguably better approximates real world data.
3) **MaxPooling:** Pooling is technique that both reduces the computational complexity of our model as well as reduces the spatial dimensions of the input volume for the next Convolutional Layer.

These Convolutional Layers can then be stacked on top of each other to produce a more complex model as required.

At the very end comes the Classification phase, which is generally a fully connected Multi Layer perceptron that uses a softmax activation function in the output layer (other classifiers like SVM can also be used).

[1]https://github.com/llSourcell/Convolutional_neural_network/blob/master/convolutional_network_tutorial.ipynb

III. Implementation

In this section we provide all the relevant steps that were taken in order to implement the methods outlined in Section II. Specifically; what technologies were used, how the data was structured and how each of the stages were trained.

A. What Was Used?

Fortunately we live in an age were most of the complexity surrounding the implementation of these methods can be abstracted away with an import statement. Below lists the key technologies used to achieve the results in section (IV).

- The code was written in **python3** using the **spyder** IDE.
- A combination of **openCV, PIL** and **skimage** was used for image manipulation.
- The **dlib** toolkit (for python) by Davis King was used to build and train the HOG+SVM object detector.
- **Keras** with a **TensorFlow** back end was used to build and train the Convolutional Neural Network.
- **Google's Cloud Platform** was used to train the CNN

B. About The Data

The input dataset for this project consisted of 207 .jpg images. In order to be able to train as well as test weather our implementation is actually working we first had to split the original dataset into 3 separate datasets, a test set, training set and validation set. The test set consisted of about 15% of images while the training set consisted of about 65% of the images. The validation set was made up of 20% of the images and was kept away during the training of both stages. Further more, some of my own images (12 of them) were added into the training dataset to increase the size of the dataset and help prevent over fitting.

After processes describes in Subsection D. Each coin in an image could then belong to 1 of 6 classes:

- ”1R”: If it was determined that the coin in focus was a one rand coin.
- ”2R”: If it was determined that the coin in focus was a two rand coin.
- ”5R”: If it was determined that the coin in focus was a five rand coin.
- ”10C”: If it was determined that the coin in focus was a ten cent coin.
- ”20C”: If it was determined that the coin in focus was a twenty cent coin.
- ”50C”: If it was determined that the coin in focus was a fifty cent coin.

C. Training The Coin Detector

Creating a HOG+SVM object detector from scratch is an challenging and extremely time consuming process. Thankfully dlib provides a simple API to do just that!

All that's required is a set of positively (and optionally negatively) annotated training images. Annotating an image simply means drawing a rectangle around the object in an image you want the detector to learn to detect.

So to train dlib's object detector to detect coins, we took the set of training images (all 143 of them) and manually drew rectangles around each coin in each image (shown in Fig 3).

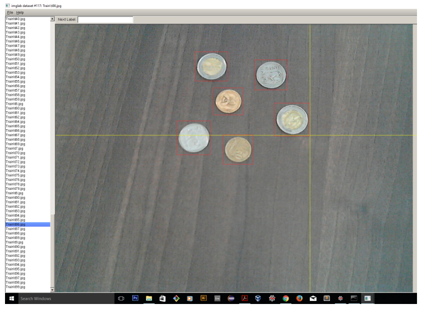
Fig. 3: Process of manually annotating coins using dlib's imglab

Approximately 2 hours and 13 hand cramps later we get an .xml file containing each image in our training set along with co-ordinates specifying the positions of each coin in an image.

This .xml file is then used to train the coin detector. Fig 4 shows the learned HOG filter.

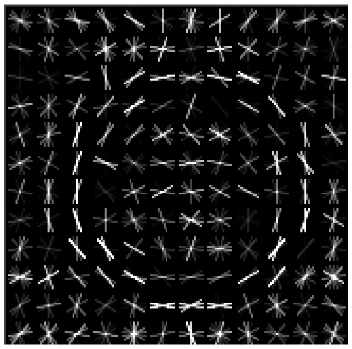
Fig. 4: Learned HOG filter from training data

D. *Training The Coin Classifier*

Training the convolutional neural network was a much more involved process, as a large amount of labeled training images of individual coins had to be gathered.

The accumulation of these images were done in a few ways.

- First the 105 labeled images provided were used as masks against the corresponding 105 original images in order to crop out each coin in the images. After that process was complete, each coin was then labeled individually according to it's actual value.(It should be noted that some coins were extremely difficult to distinguish between)
- Next, the learned coin detector described above, was run on both the training and test image set. Which then detected, segmented and saved each coin as it's own image. And again, after this process was complete, each coin was labeled individually according to it's actual value.
- After these processes were complete, openCV was used to combine all images of coins that belonged to the same class as well as to resize the images to 100x100. This resulted in approximately 2000 labeled images of individual coins. However, the distribution of images per class was extremely uneven. The "1R" class consisted of only 44 images while the "50C" class had around 600 images.
- To deal with uneven class distribution and too increase the overall size of the training data set, a technique known as **image augmentation** was performed. This not only allowed us to evenly distribute the number of images per class, but it also allowed us to drastically increase the size of our training data set. After this process was complete, we went from having around 2 000 images in total to a whopping 60 000 images, with roughly 10 000 images per class.

1) *Structure Of The CNN:*

Unfortunately, there still isn't a a definite way to determine the optimal structure and hyper parameters of a CNN. Thus, the structure and hyper-parameters chosen for this network were done so by trial and error by testing a few different structures until the final one was decided upon based on accuracy.

For this project, we settled on a 4 layer Convolutional Neural Network with the following hyper-parameters:

- Each layer took in a 100x100 RGB image and performed a 32 3x3 convolutions on the image. The outputs from that stage was activated using the RELU activation function which was then was then reduced in size using a 2x2 MaxPooling window.
- A dropout of **0.4** was used to help prevent over fitting.
- A fully connected layer of **128** neurons.
- A fully connected layer of **6** neurons corresponding to each class.
- Finally the softmax activation function was used to obtain the class probabilities.

IV. Results and Analysis

This section details the results obtained from following the implementation laid out in Section III. The first two subsections provide a brief analysis of performance and accuracy of the both stages. This is then followed by results of the solution as a whole which were obtained from running the learned detector and classifier on the set of previously unseen validation images.

A. *Testing The Coin Detector*

After training the coin detector seemed to work like magic. It's able to detect coins independent of the surface it's lying on, whether or not coins are touching, it's even able to deal with perspective! Figure 5 shows the coin detector running on a few unseen images.

After training, the detector was tested on both the training and test sets and the following performance metrics were observed:

- **Training Set Metrics:**

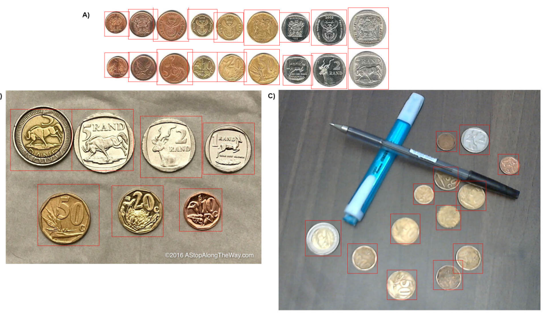

Fig. 5: Results of running the trained coin detector on unseen images. Red box indicates positive detection. Images **A) and B)** were obtained from Google, while **C)** was taken randomly from the testing set

- precision: 0.998936
- recall: 1
- average precision: 1

- **Testing Set Metrics:**
  - precision: 1
  - recall: 1
  - average precision: 1

B. *Testing The Coin Classifier*

Training the 4 Layer CNN with a data set of about 60 000 images (with a 80/20 test train split) took around 24 hours to complete. Luckily, Google's cloud computing platform allowed for the training of the network to be done on a remote virtual machine instance.

Once training was complete, the model was evaluated on the test set (which consisted of around 12 000 images) and was able to achieve the following metrics:

- **Testing Set Metrics**
  - Accuracy: 0.8325
  - Loss: 0.41785

Furthermore, to be able to visually determine the predicted class for a particular coin in an image, we used the following colour coding technique:

- Coins predicted as ”5R” are surrounded in a **RED** rectangle.
- Coins predicted as ”2R” are surrounded in a **YELLOW** rectangle.
- Coins predicted as ”1R” are surrounded in a **BLUE** rectangle.
- Coins predicted as ”50C” are surrounded in a **GREEN** rectangle.
- Coins predicted as ”20C” are surrounded in a **PURPLE** rectangle.
- Coins predicted as ”10C” are surrounded in a **BLACK** rectangle.

Fig 6 shows this colour code in action by running the learned coin classifier running on an unseen image from the validation set.

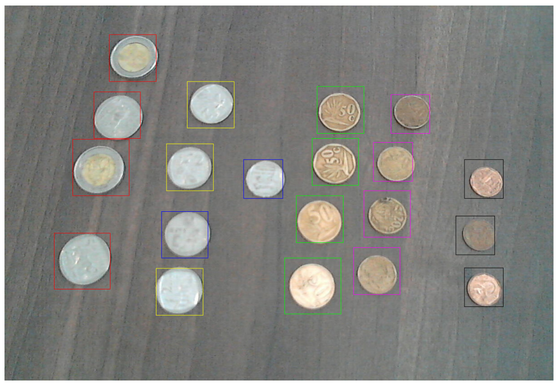

Fig. 6: Results of running the trained CNN classifier on an unseen image. It's interesting to note the single misclassification, which is actually a R2 but the system saw it as a R1

C. *Testing The Implementation*

Here we focus on testing the project as a whole from start to end. ie. Given an image we should be able to detect all coins using the trained coin detector and then classify those detected coins using the trained CNN which outputs the value of each coin as a float and then sums them up to get out the total value of the input image. The overview of this is shown in Figure 7

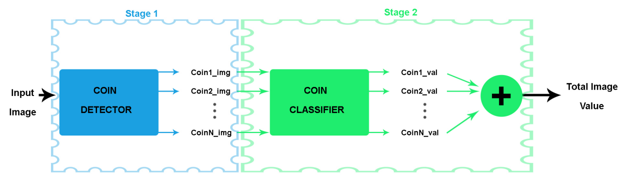

Fig. 7: Overview of implementation

To Test the implementation we ran the set of (41) unseen validation images. And in order to measure the performance of the implementation, we simply ran each image through the system and checked if the predicted output matched the actual output value.

After doing this, we observe that the system is only able to correctly classify around 13 out of the 41 validation images. While this may not be impressive, it should be noted that images that most of the coins in an image were correctly labeled, with just one or two coins being mistaken for coins from a similar looking class.

By far the biggest confusion lied between the 20C class and the 50C class. Which accounted for nearly half of all misclassifications.

V. Conclusion

In this report we have demonstrated the power that these RNN models have over sequential and/or variable length input data. In particular, we implemented a single layer RNN which takes as input a chunk of raw text and builds up a probabilistic language model to generate similar sequences of text.

Although the model that we chose to implement may look very simplistic in the context of the larger class of RNNs.

Our trained model still managed to produce desirable melodic output.

However, these models are not without their drawbacks. One of the main motivations established for using RNNs mentioned in section (I), is their potential to relate previous information to a given current task under consideration. But in some cases where more information is required, it becomes entirely possible for this 'information gap' between the relevant information and the point where it is needed to become very large. In theory, RNNs are absolutely capable of handling these long-term dependencies. but unfortunately, as that information gap grows, RNNs become unable to learn to connect the related information between time steps. This problem, commonly known as the 'vanishing gradient' problem was explored in depth by Bengio, et al. in [2], who found some fundamental reasons why it might be difficult to learn these long term dependencies using ordinary gradient descent.

Fortunately, these difficulties associated with training RNNs are much better understood[3]. Long Short Term Memory networks (LSTMs) are a special kind of RNN, which were explicitly designed to avoid the long-term dependency problem. They were introduced by Hochreiter et al. in [4], and were later refined and popularized by many researchers because of their ability to perform well on a variety of problems.

ACKNOWLEDGMENTS

Andrej Karpathy's blog post in which he demonstrated the effectiveness of a character-level language models based on RNNs.

The abc music project developed by Chris Walshaw, which made converting between a plain text file to a midi file and vice versa an extremely simple task.

The Nottingham Music Database maintained by: James Allwright and Eric Foxley which contains over 1000 Folk Tunes stored in the abc format.

REFERENCES

[1] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*, ser. CVPR '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 886–893. [Online]. Available: http://dx.doi.org/10.1109/CVPR.2005.177

[2] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *Trans. Neur. Netw.*, vol. 5, no. 2, pp. 157–166, Mar. 1994. [Online]. Available: http://dx.doi.org/10.1109/72.279181

[3] R. Pascanu, T. Mikolov, and Y. Bengio, "Understanding the exploding gradient problem," *CoRR*, vol. abs/1211.5063, 2012. [Online]. Available: http://arxiv.org/abs/1211.5063

[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735