Deep Learning Final Project

Hunter Jensen

Sean Richens

Rey Johnson

Problem

Our problem is straightforward: given an image containing a trading card, we want to identify the four corners of the card.

In the standard approach, bounding box classification returns the coordinates for a center point, as well as a width and height. In our case, this won't work because most of the images we have are shot from a perspective that's not straight-on. We need a system which will output the coordinates of each corner, allowing us to identify the bounding box regardless of the tilt.

While the motivation for this project stemmed from a personal project of one of our group members, we can see that this problem could be easily generalized. A document scanning app for a smart phone may need to identify a document and determine the bounding box to crop it correctly. The practicality of this problem makes it something worth solving.

This problem is a good candidate for deep learning because images in the data set are shot from a variety of perspectives. Lighting conditions differ dramatically across images, which can confuse standard computer vision techniques such as Canny edge detection and Hough transforms.

We began this project anticipating that a deep learning model can overcome these problems and create a better solution. Our results show that this is indeed the case.

Data Set

One of our group members had an existing set of images of trading cards for a personal project, with each image 225 by 225 pixels. In order to use this as a data set for the project, we needed to have them labeled. We built a Python application which allowed us to manually label the data set.

The program would select an unlabeled image from the data set, rescale it to be the appropriate size we needed, and display it. The user performing the labeling would then click on all the corners, starting in the bottom left and moving clockwise. After selecting all four points, the program would display the bounding box that had been manually drawn. To save the results and advance to the next image, the user would then push the "Enter" key.

A screenshot of this classification program is included below.

Initial Approaches

Our initial brainstorming included three possible approaches: an image segmentation network, a convolutional architecture with a differentiable spatial to numerical transform (DSTN) layer, and a variety of the You Only Look Once (YOLO) convolutional architecture.

After early success with the DSTN architecture, we abandoned the YOLO approach as it would be redundant. We did complete an image segmentation approach to solving the problem. The pros and cons of each approach will be discussed further in this report.

DSTN

A differentiable spatial to numerical transform layer allows a convolutional network to directly output coordinates and does so in a manner which is fully differentiable. In contrast, several other methods for detecting bounding boxes with convolutional networks rely on non-differentiable operations, such as ARGMAX or selecting the brightest point on a heatmap and then feeding it into a fully-connected layer.

Having a fully differentiable layer which directly outputs the values we want allows us to train an end-to-end network, which can backpropagate from the correct solution through the entire network, which we strongly prefer.

Our DSTN architecture adds a DSTN layer to the end of a ResNet. Instead of using a pre-trained ResNet, we decided to train our own, adapting code from a previous homework. After testing several configurations, we settled on an architecture (referred to as 'resnet6kD' in the code) with one residual block with an input size of 8 and an output size of 16 followed by a residual block with an input size of 16 and an output size of 32. We also found that a large kernel size (~15) worked well for this architecture.

In our DSTN approach to solving the problem was divided into training, test, and validation data sets. We report results on the validation data set, which is made of images which were hidden from the network during training.

Results

Yellow Borders Only, No Rotation

For our first stage, we restricted our data set to only a certain type of card which has a yellow border. We hypothesized that it would be easier for the network to detect these types of cards. This was correct, as demonstrated in the results below.

Max Distance (pixels from target)	5	4	3	2	1	0
Accuracy	99.01%	98.51%	98.02%	94.55%	68.81%	0.99%

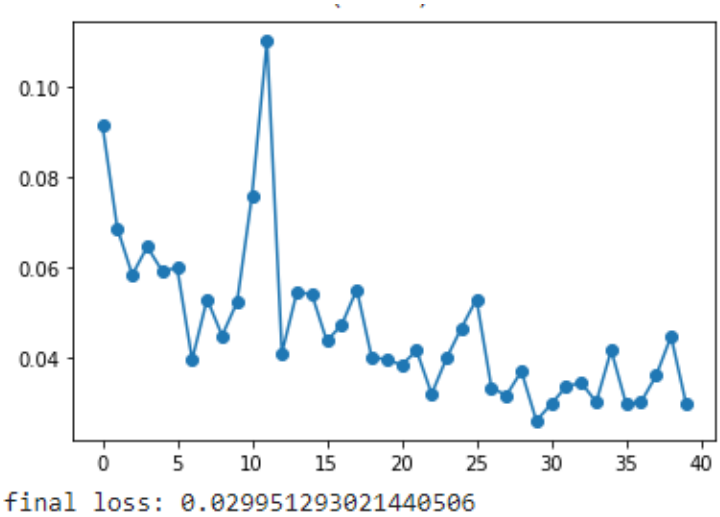
We report accuracy at several thresholds. Each threshold is a maximum distance from the target coordinates which we consider acceptable. Predictably, as this threshold tightens, the accuracy decreases. When we accept points up to 5 pixels away from the target as accurate, we get nearly perfect results. On the other hand, if we expect the network to exactly replicate the target points, we only get 1% accuracy. Considering each image is 225 pixels tall and wide, being within 5 pixels quite reasonable, especially considering that the hand-labeled points certainly aren't perfect either.

Yellow Borders Only, With Rotation

Next, we decided to expand the data set by rotating each image three times by 90 degrees. This increases the size of the dataset four-fold, and we thought that since corner detection is independent of orientation, this could make our model more generalizable. Our results after adding rotation to some of the images is reported below.

Max Distance (pixels from target)	5	4	3	2	1	0
Accuracy	97.16%	96.54%	95.43%	91.73%	62.22%	1.23%

Below is a graph of the loss during training at various epochs.

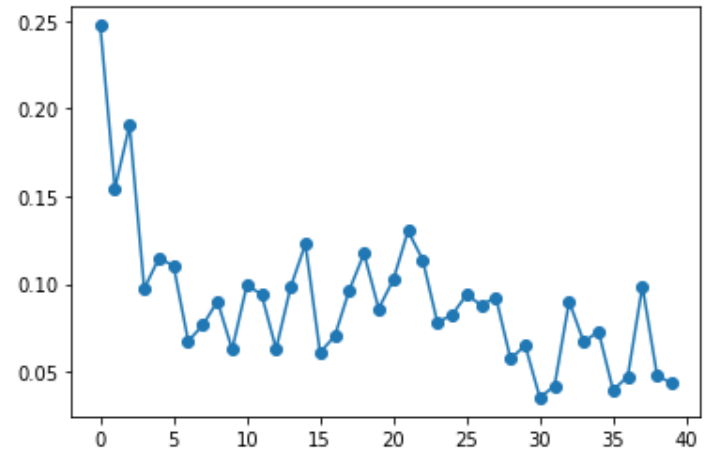


Multiple Borders, With Rotation

After we got successful results with yellow-bordered cards, we expanded the data set once again. This time, we went back and labeled more images to include multiple types of cards with various borders, making the task more generalizable. We knew this task would be harder, since the network would not be able to use the yellow border to determine the bounding box. Additionally, cards with black borders were more prone to blending in if the background was dark, also making the task harder. Our results are reported below.

Max Distance (pixels from target)	5	4	3	2	1	0
Accuracy	89.51%	85.58%	79.81%	70.28%	40.65%	0.79%

Below is a graph of the loss during training at various epochs.




```
final loss: 0.043317366391420364
```

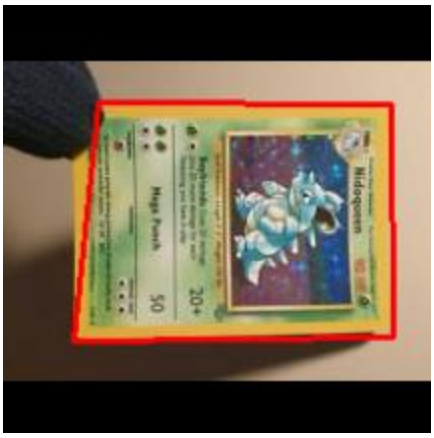
Discussion

Our results for a DSTN network met our expectations. As we had anticipated, the network did best when the data set was restrained to only images with a yellow border and a similar orientation. As we expanded the data set to include rotation, and then additional border types, the accuracy` decreased.

However, we achieved results of 89.51% with a max distance of 5 on our most challenging data set. We are pleased with this outcome.

Throughout the process, we sometimes looked at images which the network performed poorly on to find patterns in the failure. We will discuss some of these patterns that we found.

First, we found that some of our images include a person's thumb covering one of the corners. The network consistently struggles in these scenarios, because it does not know how to infer the location of the obscured corner.



We also found an instance where a digital trading card had crept into our data set, instead of an image of a physical card. Predictably, the network also struggled with this one.

Finally, we found that blurriness and harsh lighting conditions consistently led to poor outcomes. This area is a possible avenue for future research, as this problem would likely have to be solved in any real-world application for this type of system.

Image Segmentation

The other network that we successfully implemented was an Image Segmentation network based on SegNet. We hoped to address some of the issues seen above in coordinate regression, such as how obscured corners cannot be detected by our DSNT network. The goal with image segmentation was that it would be able to detect the entire card, not just the corners.

SegNet, at its core, is a combination of an encoder and a decoder. The encoder portion trains to detect what objects are in the picture, and the decoder portion figures out which pixels are a part of what object. The encoder portion of the original SegNet is the convolutional layers of a VGG16 net trained on ImageNet. Each block in the encoder has a set of convolutional layers, batch norms, and ReLUs, before ending with a max-pooling layer. The structure of the decoder portion of the net is essentially a reflection of the encoder, which allows the up-sampling layers to use the indices of their corresponding max-pooling layer.

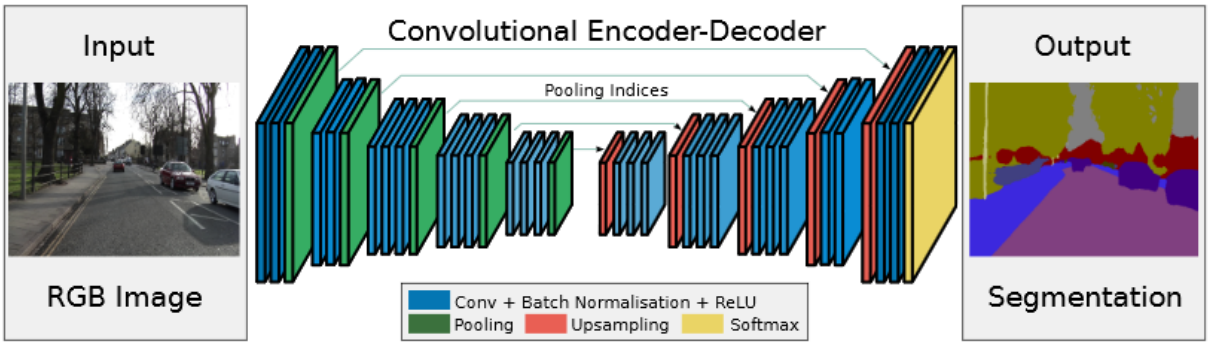


Figure 1. An overview of the SegNet network architecture from: Vijay Badrinarayanan and Alex Kendall and Roberto Cipolla. (2015, Nov 02). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. Retrieved December 11, 2020, from https://arxiv.org

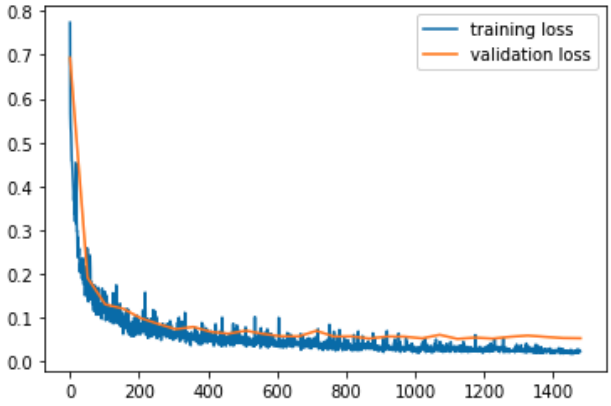
Our version of the SegNet uses a similar structure. Since we're training our segmentation net on cards instead of ImageNet objects, we don't use any pretrained parameters. We also had to make the net smaller than VGG16 due to memory concerns, however it's possible that making our net the same size as the original SegNet wouldn't have helped its performance (see below).

In order to reach our end goal of creating 4 coordinates, one for each corner, we fed the output of the segmentation into a handmade Python method to find the smallest quadrilateral enclosing the space. The function achieved this by starting with a basic bounding box, and then using a rotating caliper-based algorithm to shrink the polygon down.

The same training/test/validation datasets were used as in the DSNT model.

Results

We were able to train the image segmentation itself to have a very low loss and high accuracy, topping out around 98% overall pixel accuracy. We trained and tested different versions of our net: changing the number of parameters and convolutional layers, switching between Adam and SGD optimizers, and using different batch sizes. However, they all behaved in a similar way, none of them noticeably better or worse than the others.



Once we began running the segmented images through our bounding box function and looking at the 4 corner points it produces, we saw a drop in accuracy compared to DSNT:

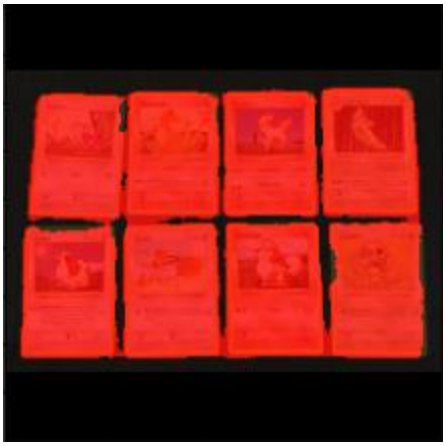
Max Distance (pixels from target)	5	4	3	2	1	0
Accuracy	28%	24%	19.5%	18%	17%	15.5%

Discussion

Our results for image segmentation didn't quite meet our expectations on accuracy, but it did show promise at what is potentially a better long-term solution. The corner point predictions were limited by noise in the segmentation network's maps that it generated. Because it was prone to creating splotches of mis-classified pixels, this made it difficult to get each corner of the card right when using our bounding box function. Shown below is an example of one case where that occurred on the top left and bottom right corners:


One thing that we could modify to fix this, if we had more time, is the loss function. Our network currently converged to a very low loss, but that loss was calculated based on how many pixels it got correct. For most pixels, however, we don't care about whether they are correct. It's only the pixels on the edge of the segmentation that affect the final bounding box. If there was a way we could modify the loss function to understand this, or to encourage the model to be more cautious about labeling pixels as positive, we think performance could improve dramatically.

Image segmentation also shows promise in its ability to detect the entire card rather than just the corners. We think with more resources, time, and examples, it would be possible to develop a highly-accurate segmentation network that would be able to handle cases our DSNT algorithm cannot. This approach also offers a benefit over DSNT in that it has the potential to classify multiple images. While DSNT does not support multiple object detection like some other non-differentiable coordinate regressors, the image segmentation approach is fully capable of multiple card detection. In practice, with our network we found that the only thing limiting this was when cards are overlapping in the image, or when splotches created connect multiple cards, causing the bounding box function to create a single bounding box for multiple cards. These are cases that we think could be eliminated with more refinement of the network. Below is one multiple-card example our network produced.



Overall, we were happy to see the results we could achieve with image segmentation, and while DSNT performs better, it was still a valuable solution to explore and shows promise at even better performance.

Conclusions

Ultimately, the best approach depends on the specific use case. As so often seems to happen in machine learning, we find ourselves answering the question of what approach is better with “it depends”.

Our DSTN network demonstrated high accuracy in finding a bounding box for a single card in an image. For an application like a document scanner app on a mobile phone, we would want high accuracy and only ever expect a single card to show up in the frame. In this case, the DSTN network is probably the best approach because it meets the needs of the use case.

In contrast, our segmentation network was less accurate at determining the specific bounding box but was able to detect multiple cards in an image. If we had a use case where we wanted to count the cards within an image, then the segmentation approach would prove superior.

Attached Documents

Submitted along with this report are several documents.

We’ve submitted Jupyter notebooks from the DSTN approach showing results in two scenarios:

- “Corner Detection – With Yellow Border” walks through the case where we restricted the training set to only cards with a yellow border, but include rotation.
- “Corner Detection – Final” walks through our final case, where we include cards with all borders and all rotations. This notebook takes a significant amount of memory to run and we had to run it on a “high memory” instance with Colab Pro.

We’ve also included a notebook with the image segmentation approach.

Works Cited

He, K., Zhang, X., Ren, S., & Sun, J. (2015, December 10). Deep Residual Learning for Image Recognition. Retrieved November 06, 2020, from https://arxiv.org/abs/1512.03385

Kocabas, M., Karagoz, S., & Akbas, E. (1970, January 01). MultiPoseNet: Fast Multi-Person Pose Estimation using Pose Residual Network. Retrieved November 06, 2020, from

https://openaccess.thecvf.com/content_ECCV_2018/html/Muhammed_Kocabas_MultiPoseNet_Fast_Multi-Person_ECCV_2018_paper.html

Nibali, A., He, Z., Morgan, S., & Prendergast, L. (2018, May 03). Numerical Coordinate Regression with Convolutional Neural Networks. Retrieved November 06, 2020, from https://arxiv.org/abs/1801.07372

Vijay Badrinarayanan and Alex Kendall and Roberto Cipolla. (2015, Nov 02). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. Retrieved December 11, 2020, from https://arxiv.org/abs/1511.00561

Appendix 1: Full ResNet Architecture

```
ResNet(
  (cnn): Sequential(
    (0): ResNetStem(
      (net): Sequential(
        (0): Conv2d(3, 8, kernel_size=(15, 15), stride=(1, 1), padding=(1, 1))
        (1): ReLU()
      )
    )
    (1): ResNetStage(
      (net): Sequential(
        (0): ResidualBlock(
          (block): PlainBlock(
            (net): Sequential(
              (0): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (1): ReLU()
              (2): Conv2d(8, 12, kernel_size=(15, 15), stride=(2, 2), padding=(7, 7))
              (3): BatchNorm2d(12, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
              (4): ReLU()
              (5): Conv2d(12, 16, kernel_size=(15, 15), stride=(1, 1), padding=(7, 7))
            )
          )
          (shortcut): Sequential(
            (0): Conv2d(8, 16, kernel_size=(1, 1), stride=(2, 2))
          )
        )
      )
    )
    (2): ResNetStage(
```

```
    (net): Sequential(
      (0): ResidualBlock(
        (block): PlainBlock(
          (net): Sequential(
            (0): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (1): ReLU()
            (2): Conv2d(16, 24, kernel_size=(17, 17), stride=(1, 1), padding=(8, 8))
            (3): BatchNorm2d(24, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
            (4): ReLU()
            (5): Conv2d(24, 32, kernel_size=(17, 17), stride=(1, 1), padding=(8, 8))
          )
        )
        (shortcut): Sequential(
          (0): Conv2d(16, 32, kernel_size=(1, 1), stride=(1, 1))
        )
      )
    )
  )
  (hm_conv): Conv2d(32, 4, kernel_size=(1, 1), stride=(1, 1), bias=False)
)
```