The Effect of Graph Dependencies on Multi-Object Classification

Jordan Levy
University of California, San Diego
La Jolla, California
jdlevy@ucsd.edu

Abstract

In this paper, we isolate the effect of training deep learning networks on graphs versus multi-layer perceptrons, after extracting features from a CNN. We propose 4 unique architectures designed to isolate the effect graph neural networks will have on training our model. Our results are inconclusive in that we did not say definitively which models outperformed another, but we were able to produce training loss curves that show the potential for Graph Neural Networks, as well as visualizations that again show the potential for GNNs in the future. We draw attention-weighted graphs over images to show that GNNs can be effective in modeling relationships between objects within an image.

1 Introduction

In any deep learning problem, we wish to best model the dependencies between our input and output. Over the past several years, the field has seen an increase in the application of attention mechanisms and graph neural networks to better understand the structure of our input data and how it relates to our output. In this paper, we compare a traditionally built CNN feature extractor into Multi-Layer Perceptron (MLP) against a CNN into a Graph Convolution Network for multi-object classification. We use the COCO-2017 dataset (Microsoft Corporation, n.d.), which contains over 100,000 training images and 5,000 validation images, to perform multi-object classification. The COCO dataset has numerous segmented objects in each image, and we will be classifying them into one of 80 different categories.

What this project aims to assess is the influence of Graph Neural Networks and, more broadly speaking, the influence of adding dependencies between objects in achieving a more robust classification model than if we were to treat each object as independent. We hypothesize that by modeling each dependency between objects via a graph that we can outperform a model that does not take into account these dependencies. As an example of a dependency, we may see a human near a coffee mug in several images, so the link between human and coffee mug may be very strong, as each are more likely to appear around each other. We will be exploring this concept in greater detail, as we observe the relations in our data.

2 Methods

2.1 ETL

As mentioned above, we are using the COCO-2017 dataset to conduct our analyses. COCO, which stands for Common Objects in Context, is primarily used for object detection and segmentation, though there are a wide variety of tasks that can be done with the COCO dataset. Here we are using COCO to classify multiple objects in the same image, a task also referred to as multi-object classification. In order to do this, we first load in all the training data as well as training annotations. COCO provides an API to interact with the dataset, a series of high level functions that allow us to easily get access to and load image and annotation data into our jupyter notebook (GitHub, 2020).

An important next step in this process is to load our data into pytorch, the Python module we will be using for deep learning inference. For the purpose of this project, which is to classify individual objects within an image, we decide to crop each image into a set of subimages, with each subimage defined by a bounding box given in the image's annotation. The following figure gives an illustration as to how this looks:

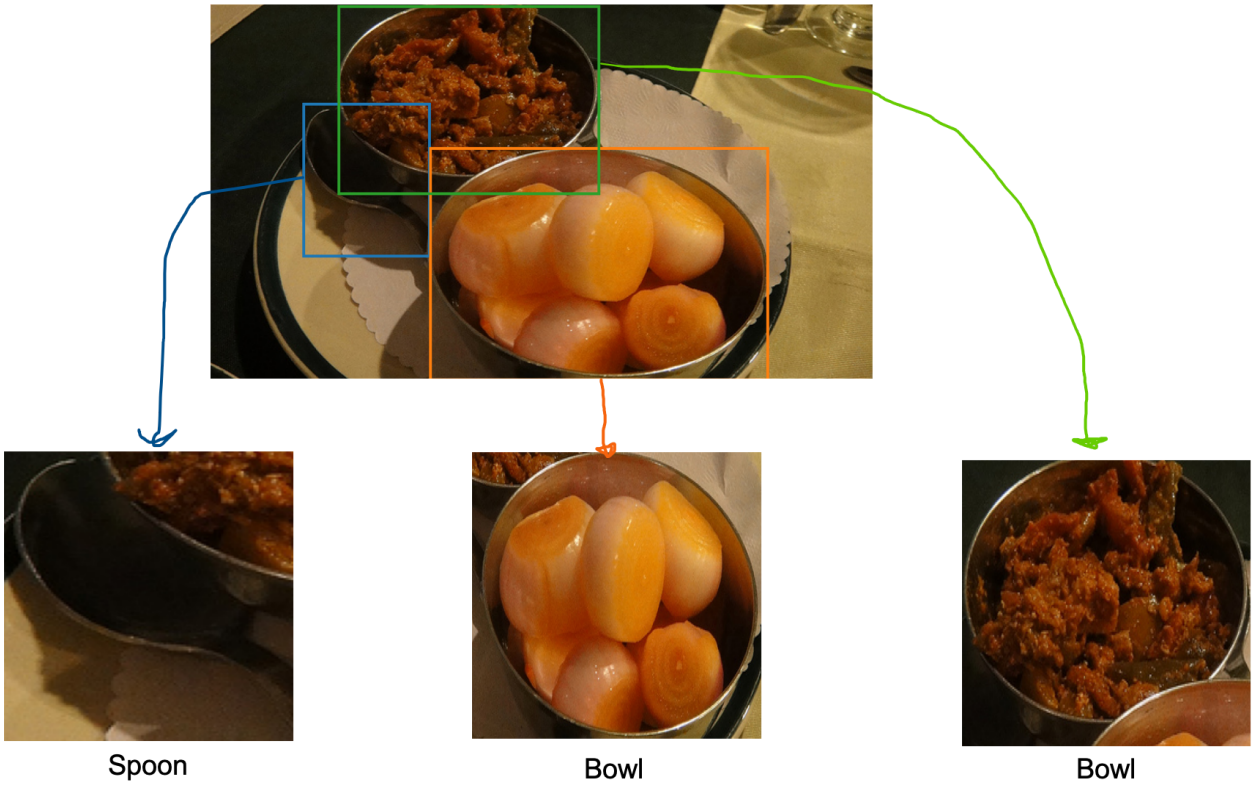

Figure 1: An example of extracting three subimages from a COCO image with bounding boxes

As seen in the figure above, we are extracting multiple subimages from each COCO image based on their bounding boxes, then resizing each sub-image to 128x128 pixels. This allows us to pass in our data to pytorch, as all images now have an equal shape, and begin training our first model. Before getting into the various architectures we compare in this project, let's look at the second half of the ETL process, which is generating the graphs from our COCO dataset.

Generating graphs for each training and validation image turned out to be one of the more challenging tasks in this project, as there are many ways in which one can go about graph creation. I chose to create a KNN graph with a value of 3 for k. This means that each object will be connected to the 3 nearest objects and vice versa, where the closest objects are determined by the xy values that make up the center of each sub-image. Once this graph is produced, we can then go on to training our model.

2.2 Network Architectures

We will be comparing 4 separate deep learning architectures:

1) Pre-trained Resnet Feature Extractor with a fine tuned MLP
2) Pre-trained Resnet Feature Extractor with a fine tuned GCN
3) COCO-trained on top of Resnet, End-to-End Learning with MLP
4) COCO-trained on top of Resnet with a fine tuned GCN

For the sake of consistency, all of these models were trained 10 epochs, a learning rate of 1e-4, and a batch size of 12; though changing these parameters is certainly something to look into. The following figure describes the basic architecture and workflow of Model 1:

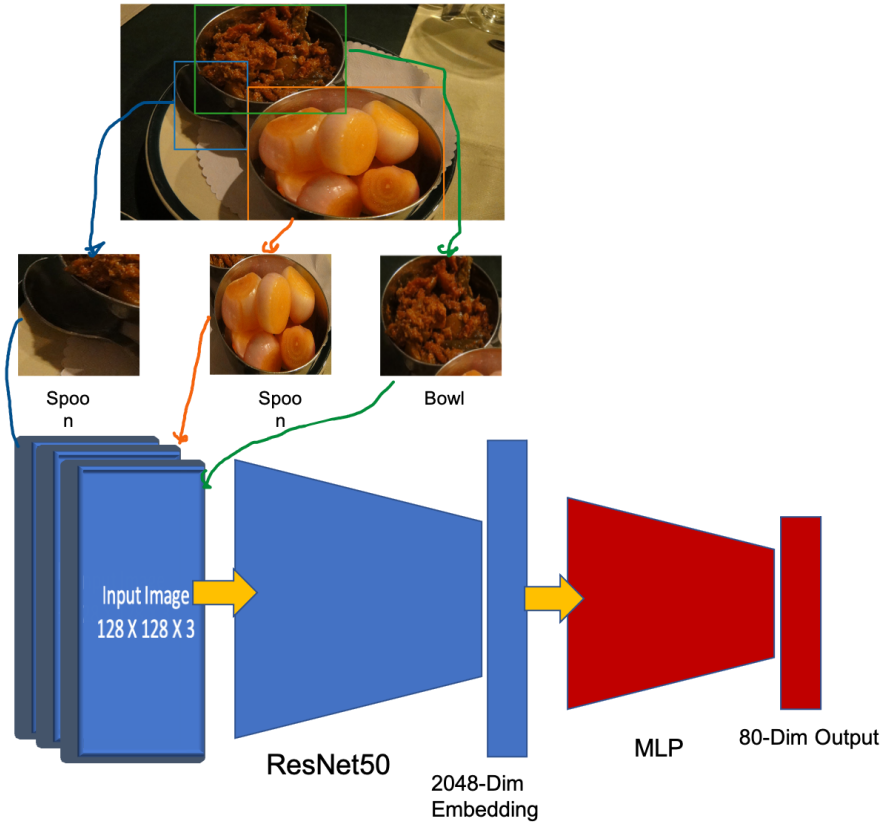

Figure 2: Model 1 Architecture

As the figure above indicates, our model is essentially utilizing two separate models, the pretrained ResNet-50 model, and our fine-tuned MLP that requires training on the COCO

dataset. Here, Model 1 can be thought of as domain maladapted and independent, as we are simply using pre-trained Resnet features from our CNN and are pushing it through our fine-tuned MLP. We are not reliant on COCO for feature extraction, and the model treats all images as independent due to a lack of graph structure.

This leads us into Model 2, which looks fairly the same as Model 1, but with a GCN applied on the 2048-Dimension embedding instead of an MLP. The idea here is to observe the effect of adding dependencies between our objects on the performance of our model.

Models 3 and 4 are essentially the same, but trained on COCO during feature extraction instead of using ResNet50 weights. However, it is important to note that Model 4 does not use the same end-to-end learning that Model 3 does, rather it simply takes the feature extractions from Model 3 and applies them to a GCN. With more time, we would have created a Model 5 that has end-to-end learning with a GCN.

4 Results

Now that we have our architectures defined and our data loaded into Pytorch, we can train our models and observe the results. While I was able to train multiple models, I had difficulties in sticking to the proper architecture and ultimately have run out of time to produce validation accuracy metrics of the models I've trained. However, I was able to implement some interesting visualizations that give us insight into what the GCN found important.

Before getting to that, I will show the training loss curves between Model 1 and 2, and Model 3 and 4, respectively. Note that in the figures below, Models 1 and 3, the MLP models, are in blue while Models 2 and 4, the GCN models, are in orange.

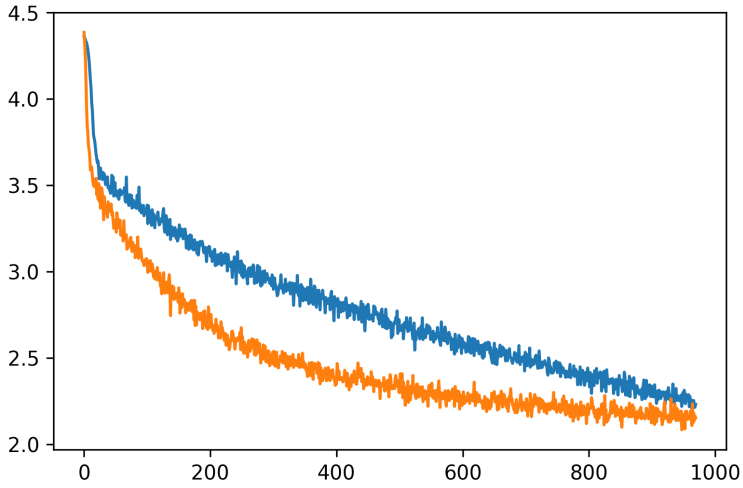

Figure 3: Training loss curves between Model 1 (blue) and Model 2 (orange)

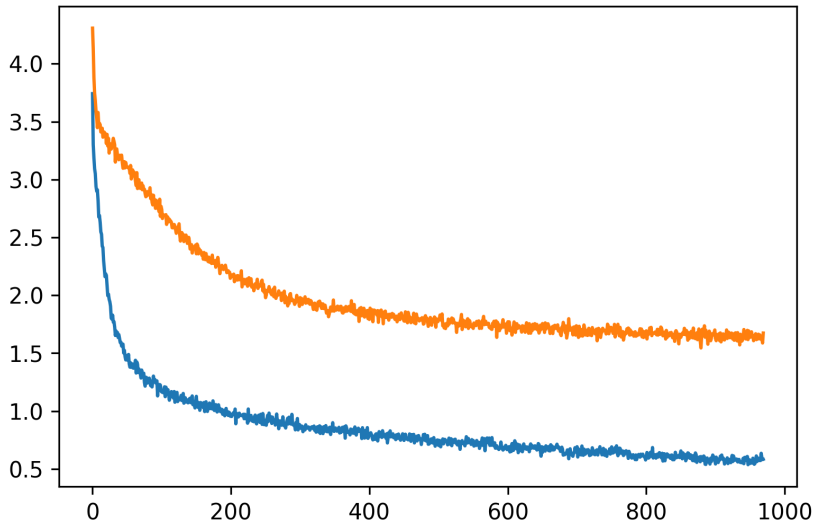

Figure 4: Training Loss Curves between Model 3 (blue) and Model 4 (orange)

There are several takeaways to be had by looking at the figures above. First, based exclusively on training loss, Model 2 appears to outperform Model 1, though both models appear to have room for improvement with future epochs. Second, Model 3 appears to outperform Model 4, somewhat of a reversal of Figure 3, as Models 1 and 3 are MLP based whereas Models 2 and 4 are GCN based. However, upon further inspection, it makes sense that Model 4 would outperform Model 3, as Model 3 had end-to-end learning whereas Model 4 simply used Model 3's feature extractions before beginning to train. So while our results are inconclusive based on the information provided, it does seem like the GCN has the capability to outperform the MLP when on a level playing field, as seen in Figure 3.

Tangential to the issue of accuracy, we also wanted to give our GCN model some form of explainability, to examine the magnitude of dependencies between nodes. This was made possible by the type of GCN we chose, which included an attention mechanism. This allows us to observe the attention weights in the forward pass after training our model and come up with some interesting visualizations on our validation images. Here are some examples of what we

were able to produce:

Figure 5: Example of Attention Weights drawn between objects connected in a KNN Graph

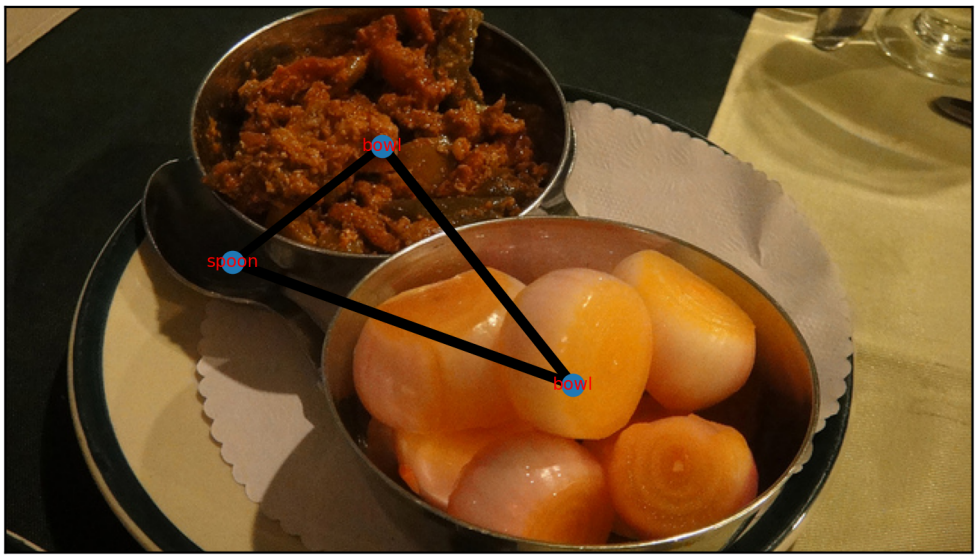

Figure 6: Another example of Attention Weights drawn between objects in a KNN Graph

In the two figures above, we see the potential but also current limitations of what can be achieved here. In Figure 5, we are able to see some meaningful differences between the edge weights/widths, as bird to person is not nearly as strong as person to person or even person to boat. However, in Figure 6, we see that this approach is not really effective when we have a few number of items in the image. We can see differences in the width of edges, but they are oftentimes hardly noticeable. Overall, we see this as a great opportunity for growth and we

expect that as our model improves, so will the attention weights in terms of explaining the relation between objects.

5 Discussion

Irrespective of the results of this endeavor, there is much more we can do to further extend this analysis. There is a lot of work to be done in fine-tuning the models that I did not get around to. Additionally, I should be keeping track of validation loss at the end of each epoch and implement early stopping, this will help with understanding at what point the model is overfitting.

Overall, I ended up falling short of what I wanted to accomplish in this project, but I plan to continue working on it and refining until I am satisfied with the results. I think there is a lot to be explored in this domain, and it is something that I hope to pursue in the future.

Another extension I would like to look into is using the Visual Genome set in a GCN. Images in Visual Genome come with a scene graph with semantic and spatial relationships provided between the objects in the image. I want to get more familiar with the ins and outs of the Visual Genome dataset and ultimately apply what I have learned in this project to that dataset and see what results I can get.

Based on my preliminary results, it seems as though Graph Neural Networks can provide a boost over a generic MLP, and I believe that with a more complex GNN I can accomplish much more, both in terms of accuracy and in explaining the relations between objects in an image, as we saw several examples of in the results.

6 Bonus Points

I believe this project could be worthy of bonus points as it seems fairly novel. I haven't seen a tutorial or repository online that does all that this project does, specifically with regards to the multi-object classification in conjunction with graph neural networks. I borrowed some code to create the dataset class for COCO, but ended up making several changes to incorporate all the labels as well as cropping and creating subimages. Additionally, the graph attention figures were all produced from code I built myself. Overall I really enjoyed working on this project and wanted to thank the teaching staff and Professor for putting this class together.

References

GitHub. (2020, February 20). *COCO Dataset*. COCO API. https://github.com/cocodataset/cocoapi

Levy, J. (2020, August 11). *GCN4R*. Github. https://github.com/jlevy44/GCN4R

Microsoft Corporation. (n.d.). *MS-COCO Dataset*. Common Objects in Context. https://cocodataset.org

Nakamura, T. (2020, April 9). *Medium*. How to train an Object Detector with your own COCO dataset in PyTorch (Common Objects in Context format). https://medium.com/fullstackai/how-to-train-an-object-detector-with-your-own-coco-dataset-in-pytorch-319e7090da5