Scene Graph to Image Translation: Graph Convolutional Network Approaches

Oghenetegiri Sido
Stanford University
osido@stanford.edu

Abstract

How can the understanding of a complex concept be tested? One test involves generating original content based on the entities and relationships in that concept. In our context, our concept is a scene graph with objects and their relationships and our final generated content will be an image depicting those objects and relationships. We process the scene graph with a graph convolutional network, generating embeddings for objects and their relations. These embeddings are passed downstream to derive a scene layout by predicting bounding boxes and segmentation masks for all objects. Then, that resultant layout is converted into an image with the cascaded refinement network, which uses discriminators to produce realistic images. We focus on the graph convolutional network module and implement several proven and novel approaches to process the scene graphs. We score the efficacy of our approaches on the COCO-Stuff and Visual Genome datasets with a variety of quantitative and qualitative metrics.

1. Introduction

With a written or verbal description of a scene, humans are adept at imagining how the scene would actually appear in real life. Can machines do this? That is our motivating question. True understanding of complex ideas is shown through the **generation** of new, sufficiently complex material. Text to image translation is an extremely popular and well-researched problem. We've seen tremendous advances in that domain. Nevertheless, the text to image translation problem is fairly constrained to simple sentences that lack rich syntactical structure and complex linguistic dependencies. Notwithstanding, graphs are extremely well suited to rich, branch-like structures like complex sentences. Accordingly, we will focus on the scene graph to image translation, which is a new and challenging problem. We will focus primarily on techniques to generate rich object and relationship embeddings that can boost the quality of generated images downstream. To accomplish this, we propose a series of methods. All of which are inspired from a baseline, which comes from [3]. This paper, [3], uses a multi-layered Graph Convolutional Network to generate neighborhood-aware embeddings for nodes and edges in the graph. In the scene graph problem, **nodes** are objects including cars, people, etc. **Edges** in this setting are relationships. For example, if a boy is holding his dog, the boy and the dog would be separate objects and the act of holding the dog is the relationship between the two objects. We implement and explore the efficacy of several approaches, including an RNN model, a Random Walk model, a Graph-Sage Max-Pool model, a Graph-Sage LSTM model, Graph-Sage Mean Aggregator model, and Graph Attention Model.

2. Problem Inputs, Outputs, and Losses

2.1. Inputs and Outputs

As **input**, we load scene graphs examples from the Visual Genome [4] and COCO [5] datasets. Our **output** will be a 64 x 64 generated image.

2.2. Losses

As detailed in [3], our model seeks to minimize box loss, mask loss, pixel loss, image adversarial loss, object adversarial loss, and auxiliary classifier loss. The majority of these losses focus on the computer vision aspect of the project, which includes predicting bounding boxes, penalizing differences in masks for the layout stage, L1 losses for differences between given ground truth images and our model generated images, and adversarial losses based on the strength of our image and object discriminators. Although our focus is on extracting value from the graph, these losses serve as indicators as to the efficacy of our various graph convolutional network methods. We know this, because the quality of the embeddings for the objects and relationships in the scene graph will have a huge impact on downstream success in the computer vision pipeline of the network. The bounding box loss, especially, serves as a robust indicator of the utility of the generated embeddings, as bounding box prediction is the first step in the pipeline after the embeddings are generated.

3. Related Work

There are several works that can be applied to our scene graph to image translation problem. We are focused on innovative, effective graph convolutional approaches that can help us generate useful embeddings from our scene graph. [6] covers various graph neural networks and their applications in the wild. These networks include state-of-the-art recurrent, convolutional, autoencoder, and spatial-temporal models. Since our focus is to generate rich embeddings for the objects and relationships in our scene graph, we found the section on Graph Convolutional Networks (GCNs) particularly compelling.

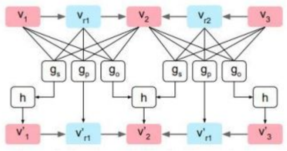


Figure 1: Pictoral representation of a single graph convolutional layer

$$\mathbf{h}_i^k = \mathrm{GRU}\left(\mathbf{h}_i^{k-1}, \sum_{v_j \in \mathcal{N}(v_i)} \mathbf{W}\mathbf{h}_j^{k-1}\right),$$

Figure 2: GRU equation for computing node representation

3.1. Graph Convolutional Networks (GCNs)

Much like 2D convolution, which is often utilized in the computer vision space for segmentation, image classification, and object detection tasks, GCNs work well to the graph domain. GCNs leverage neighborhood and eventually global information to generate node embeddings for the nodes in a given graph. On a given layer k, GCNs use the previous representations of the node and its neighboring nodes' representations from layer (k-1) to compute new representations for layer k. This layering of GCNs facilitates the *passing* of more information from nodes and edges outward to the rest of the graph [6], creating both a locally and globally-aware representation for each node and edge[1]. This is shown in Figure 1.

In [3], a spatial-based convolutional network is used where an order-invariant aggregation function is used. With regards to the aggregation functions used in [3], average and sum are employed. Graph-Sage, however, also approaches GCNs with novel variants, that we build upon [1]. These approaches focused on *generalized aggregration*, including a max-pool, min-pool, mean, and LSTM aggregators. Since max-pooling techniques have performed well in the computer vision domain in terms of gathering **key** information from feature maps, we use this architecture to pull out important information for our embeddings. We also implement the mean and LSTM Graph Sage variants.

3.2. Recurrent Graph Convolutional Networks

While the order-invariant GCNs discussed above are adept at determining a node's influence throughout the network, it has no notion of order. Additionally, depending on the number of layers and the size of the graph, they are likely to encode more local than global information for each node, limiting the expressiveness of the resultant embeddings. This provides an opportunity to leverage the order of objects with a recurrent neural network that has a stronger notion of temporality.

3.2.1 Gated Recurrent Unit

We were inspired by [2]'s recurrent approach, in which a Gated Recurrent Unit (GRU) network is applied to graphs. Their network, like spatial-convolutional networks, leverages local neighborhood information to generate node and edge embeddings. Their basic approach is detailed in Figure 2. We see that this method uses the node's previous representation and sum aggregration of neighbor embeddings that are projected into a high-dimensional space. It runs these stacked neighbor embeddings through a GRU to compute the new representation for the node. We look to expand upon this work for several reasons. GRUs have memory units that are adept at learning what to remember and what to forget. Since our downstream task is in-painting for a generated image, we need rich embeddings from the graph that can encode useful information and discard meaningless noise. Recurrent approaches, in general, have become popular for exploiting temporality and order but [2]'s GRU approach does not address the ordering of the embeddings as input to the GRU, which may not be the best approach going forward. Notwithstanding, we implement an RNN baseline, so that given a single node and all of its neighborhood nodes, we run all neighborhood nodes's representations through an RNN and use the final hidden state as the representation of the original given node. We are actively exploring other ways to leverage temporality, so that the order in which nodes are connected (objects and relationships, in our case). Currently, we have a loose understanding of temporality and sequence, as it's hard to find that in a graph.

[1]Note that in our problem, nodes are objects in the scene and edges are relationships between objects

3.2.2 Graph Sage

In [1], the authors also utilized a recurrent neural network that handles ordering. While we use an RNN, they use an LSTM and always shuffle the neighbor embeddings of a given node before running those shuffled embeddings through their LSTM. Their approach has two strengths: the use of the LSTM versus a less powerful RNN that we used in our baseline and the use of shuffling to mitigate the ordering issue. Inspired by their approach, We also implement a model inspired by GraphSage-LSTM that adopts its strengths but differs in the embeddings used. Their approach uses neighborhood embeddings from the previous layer to feed into the LSTM. We, however, project those neighborhood embeddings into a high-dimensional space before running them through the LSTM. This non-linear projection has the potential to put the embeddings in a more useful embedding space for the LSTM.

4. Data

4.1. Data Collection

Following the original paper [3] in which the scene graph to image translation problem is first proposed, we use the COCO [5] and Visual Genome [4] datasets for this project. COCO contains 40k train and 5k validation images that have associated bounding boxes and segmentation masks. We use the baseline implementation in [3] to preprocess the data, gathering annotation information to construct the scene graphs, which are the inputs to our model. Additionally, we use Visual Genome, which has 108k images that come pre-annotated with scene graphs. The model can be regularized by combining both datasets for training and evaluation.

4.2. Initial Findings and Summary Statistics

Additionally, in accordance with the data split detailed in [3], we use an 80-10-10 split for our train, validation, and test set splits, respectively. Only objects that occur in images over a certain threshold (2000) are kept in the dataset, leaving us with 62k training, 5k valuation, and 5k test examples. These images typically contain ten objects and five relationships on average [3].

5. Mathematical Background / Pseudocode

Spatial-Convolutional Baseline: the baseline implemented originally in [3] follows this format:

```
for k in range(L):
    for v in V:
        avg_vector = AGG(Q^k * u^{k-1} for u in N
    ↪ (v))
        h_{v}^{k} = MLP(avg_vector)
```

Listing 1: Spatial-Convolutional Baseline

AGG: aggregation function - average is used here
L: number of layers
V: set of nodes
Q^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

RNN Model:

```
for k in range(L):
    for v in V:
        neighbors = STACK(Q^k * u^{k-1} for u in
    ↪ N(v))
        output, last_hidden = RNN(neighbors, 0)
        h_{v}^{k} = MLP(last_hidden)
```

Listing 2: RNN Model

L: number of layers
V: set of nodes
Q^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron layer i of GCN Here, the initial hidden state (h_0), which is fed to the RNN at the first time step, is a vector of zeros, and the final hidden state is used as the encoded representation of h_n^k. Our netion of time t is linked with the index of one of neighbors of node v.

Random Walk Model:

```
for k in range(L):
    for v in V:
        neighborhood = N(v)
        augmented_neighborhood = neighborhood +
    ↪ sample_for_random_nodes(V)
        h_{v}^{k} = AGG(Q^k * u^{k-1} for u in
    ↪ augmented_neighborhood)
        h_{v}^{k} = MLP(h_{v}^{k})
```

Listing 3: Random Walk Model

AGG: aggregation function - average is used here
L: number of layers
V: set of nodes
Here, the probability of a random node in V being added to v's neighborhood is $\frac{1}{|V|}$
Q^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

Graph-Sage-MaxPool:

```
for k in range(L):
    for v in V:
        max_vector = AGG(Q^k * u^{k-1} for u in N
    ↪ (v))
        curr = RELU(W^i *  max_vector)
        prev = h_{v}^{k-1}
        curr_prev_stack = [curr, prev]
        h_{v}^{k} = RELU(Y^k * curr_prev_stack)
```

Listing 4: Graph-Sage-MaxPool

AGG: aggregation function - **max element-wise** is used here

L: number of layers
V: set of nodes
Here, the probability of a random node in V being added to v's neighborhood is $\frac{1}{|V|}$
Q^i, W^i, Y^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

Graph-Sage-LSTM:

```
for k in range(L):
    for v in V:
        neighbors = STACK(Q^k * u^{k-1} for u in
    ↪ N(v))
        neighbors = shuffle(neighbors)
        output, last_hidden = LSTM(neighbors, 0)
        curr = RELU(W^i *  last_hidden)
        prev = h_{v}^{k-1}
        curr_prev_stack = [curr, prev]
        h_{v}^{k} = RELU(Y^k * curr_prev_stack)
```

Listing 5: Graph-Sage-LSTM

AGG: aggregation function - **max element-wise** is used here
L: number of layers
V: set of nodes
Here, the probability of a random node in V being added to v's neighborhood is $\frac{1}{|V|}$
Q^i, W^i, Y^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

Graph-Sage-Mean:

```
for k in range(L):
    for v in V:
        neighbors = STACK(Q^k * u^{k-1} for u in
    ↪ N(v))
        old =  h_{v}^{k-1}
        old_neighbors_stack = [old, neighbors]
        h_{v}^{k} = MLP(AGG(old_neighbors_stack))
```

Listing 6: Graph-Sage-MaxPool

AGG: aggregation function - **mean** is used here
L: number of layers
V: set of nodes
Here, the probability of a random node in V being added to v's neighborhood is $\frac{1}{|V|}$
Q^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

Graph-Attention-Network:

```
for k in range(L):
    for v in V:
        neighbors = STACK(Q^k * u^{k-1} for u in
    ↪ N(v))
        neighbors_proj = W^k * neighbors
        old =  h_{v}^{k-1}
        old_proj = W^k * obj
        sim_vector = neighbors_proj * old_proj
        sim_vector = softmax(sim_vector)
        scaled_neighbors = neighbors (pointwise-
    ↪ mult) sim_vector
        h_{v}^{k} = MLP(AGG(scaled_neighbors))
```

Listing 7: Graph-Attention-Network

AGG: aggregation function - **sum** is used here
L: number of layers
V: set of nodes
Here, the probability of a random node in V being added to v's neighborhood is $\frac{1}{|V|}$
Q^i: learnable weight matrices for layer i of GCN
MLP: Multi-Layered-Perceptron

6. Algorithms and Approaches

The mathematical underpinnings for the following methods are discussed in the *Mathematical Background / Pseudocode* section. All approaches are discussed in terms of node v.

6.1. Hyperparameter Choices

In order to standardize results with our different model choices and the baseline, we will use the following hyperparameters: Our embedding dimension is 128, our output dimension is 128, our hidden dimension withing the Graph Convolutional Layer is 512, the number of layers for the GCN is 5, our learning rate is $1e^{-}4$, and our batch size is 32.

6.2. Spatial-Convolutional Baseline

We run the baseline, which currently uses a spatial-convolutional model. For a given node v at layer i, its representation is computed as a function of its neighborhood at layer (i - 1). Node v's neighborhood's embeddings are projected into a high dimensional space with a learned matrix (Q^i) and then aggregated via an average function. This average embedding is then run through a multi-layer perceptron to get the final encoding of v for layer i. As discussed in class, subsequent layers ($>$ i) will all have increased graph global awareness, as every i represents the number of hops and interaction that node v can have. This means as i – the index for the layer – increases, objects, which act as our nodes, will have a greater awareness of other objects in the scene. As shown in the previous section, this serves as an excellent starting point from which we can build.

6.3. RNN Model

Here, we seek to find a notion of ordering among nodes by stacking all of node v's neighbors into a larger 2D matrix. Then, we run this matrix through an RNN, where the input at each time step is a single neighbor vector while the input hidden state at every time step will be the output hidden state of the previous step. We elected to have initial hidden state (h_0) as a vector of zeros, and chose the final hidden state as the encoded representation of v. In this way, the model learns how to compute v's representation as a function of it's neighbors but also considers the order in which node v's neighbors were connected to it. The notion of order is loose here as a graph's node and edges do not relate easily to a sequential order. Hence, we do not shuffle the order of the neighbor embeddings for a semblance of order. Notwithstanding, We fully implemented this approach to determine whether there were any gains to computing v's representation in a sequential, ordered process.

6.4. Random Walk Model

This novel approach is a combination of Spatial-Convolution and the Random Walk model, popularized from the PageRank algorithm. PageRank demonstrates the utility of occasionally hopping to random page versus a linked page. This is parameterized by the probability of random jump, as a higher jump probability corresponds to more random exploration of the graph. We noticed that spatial-convolution focuses solely on aggregating neighbor's representations. This approach creates strong local representations of nodes that gradually become more globally-aware of the graph as the number of layers increases (i.e h_v^i is less globally aware than h_v^{i+1}). Accordingly, by adding random nodes to v's neighborhood, we can speed up the global exploration of the graphs, accelerating the time needed to learn a richer, globally-aware representation of v. Because we uniformly sample from all of the vertices in V, the probability of a random node in *V* being added to v's *augmented neighborhood* is $\frac{1}{|V|}$. Currently, I chose to sample 10% of the size of v's neighborhood. This is a heuristic to allow the majority of the nodes in the *augmented neighborhood* of v to come from from v's original, unmodified neighborhood. Overaggressive sampling would likely hinder creating a strong local, neighborhood-aware representation. We can test this hypothesis in future work.

6.5. Graph-Sage-MaxPool

This approach, originally detailed in [1], is an innovative approach for generating embeddings. It builds upon classic GCN approaches by projecting all of node v's neighbors into a high-dimensional space. Then, we take the pointwise-max of all of the projected neighborhood embeddings as our *max-vector*. Since this takes the max of all dimensions, we hope to select key features from all of the neighbors of v. After running *max-vector* through a linear layer, we concatenate it with h_v's representation from the previous layer. That concatenation is run through another linear layer to get the final representation for v. By directly using the previous representation of v in the concatenation with the *max-vector*, we get a *skip-connection* between the previous and current layer [1]. Thus, the network can learn how to select the relevant aspects from the v's previous representation and the aggregation at the current layer.

6.6. Graph-Sage-LSTM

As mentioned previously, our RNN model did not have a clear way to leverage order to effectively utilize a recurrent structure. However, this approach does handle order by stacking all of node v's neighbor embeddings and then **shuffling** the embeddings. This shuffling completely discards the notion of ordering by shuffling. The authors hypothesize that will train the LSTM to be "permutation invariant" by always shuffling the input embeddings. Finally, the last hidden state is concatenated with h_v's representation from the previous layer. That concatenation is run through another linear layer to get the final representation for v. This concatenation is run through another linear layer facilitating a *skip-connection* between the previous and current layer. The network will learn to pick useful information from the previous layer and the LSTM output, which can keep important old information and use new aggregated information from the LSTM.

6.7. Graph-Sage-Mean

This model is closely related to the baseline [3] in that it aggregates the neighbors in preparation for a mean aggregation. This differs, however, because it stacks the projected neighborhood embeddings and h_v's representation from the previous layer. Then, it takes the mean of those stacked embeddings and runs the resultant embedding through a multi-layer perceptron to get the final representation for v. Note, however, that the baseline differs as it does not explicitly incorporate h_v's representation from the previous layer. Also, note that the original Graph-Sage-Mean model [1] does not project the neighborhood embeddings into a high-dimensional space. We added this to model, because we hypothesize that a learned neighborhood embedding space will yield more *expressive* embeddings for use downstream.

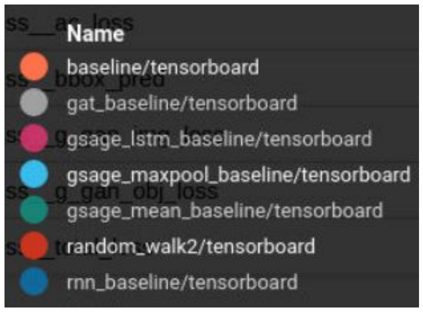
6.8. Graph-Attention-Network

As seen in class, when computing the representation for v, all prior approaches gave equal weighting to each neighbor embedding when aggregating and calculating the new representation for v. Here we compute each neighbor's importance to v as a scalar multiple. We accomplish this by projecting all neighborhood embeddings and h_v's representation from the previous layer into the same embeddings space with a learnable similarity weight matrix W_{sim}. We then dot product each projected neighbor embedding with v's projected embedding ($u^T h_v$ = sim) to derive the similarity to a particular neighbor. We softmax over all of these similarities for all neighbors to determine relative importance. Then, we sum the neighbor embeddings according to their importance ($u_1 * sim_1 + u_2 * sim_2 + ...$) and run the result through a multi-layer perceptron to get the final representation for v.

7. Tangible Contributions

Thus far, I've augmented the current baseline code with tensorboard to visualize training and validation statistics. Additionally, I implemented two novel models: the spatial-rnn model and the spatial-random-walk model. I also implemented variants of Graph-Sage-MaxPool, Graph-Sage-LSTM, Graph-Sage-Mean, and a Graph-Attention-Network.

8. Results and Findings



Name
● baseline/tensorboard
● gat_baseline/tensorboard
● gsage_lstm_baseline/tensorboard
● gsage_maxpool_baseline/tensorboard
● gsage_mean_baseline/tensorboard
● random_walk2/tensorboard
● rnn_baseline/tensorboard

Figure 3: Model Legend for Tensorboard

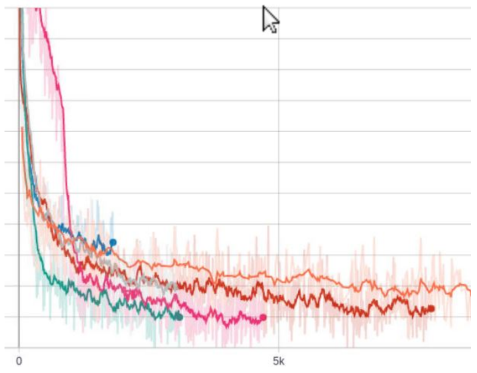


Figure 4: Training loss for various models

8.1. Total Loss

Due to time and compute constraints, some of our six models are still training, so we will present our current results [2]. Please note that the legend for the models can be found in Figure 3.

[2]Model Legend in Figure 3

8.2. Highest Performing Models

As seen in the training loss curve shown in Figure 4, several of our models outperform the baseline, which is pictured in orange. Our Graph-Sage-LSTM and Graph-Sage-Mean are the most performant in terms of minimizing our total loss. This is likely due to the incorporation of h_v's representation from the previous layer for a given node v. The other models, with the exception of Graph-Sage-MaxPool, do not use the previous layer's representation. The ability to use the aggregated information and the previous layer's representation of node v appears to allow the model to generate richer embeddings. It is intuitive that making use of previous information for node v and the current aggregated information allows the model to learn more versus the other models that rely solely on the aggregated information at the current level.

8.3. Graph-Attention-Network Performance

Our Graph-Attention-Network also performs very well, closely marking the top Graph-Sage models in total loss. Clearly, there is value in attenuating the importance of the neighborhood embeddings based on importance to a give node v. We learned this importance with a weight matrix. Since we used a very simple attention scheme, more work can be done here as well.

8.4. Worst Model

However, we see in Figure 5 that Graph-Sage-MaxPool performs extremely poorly compared to the rest of the models, sitting in last in terms of loss. Further investigation is needed into that matter, but we hypothesize that the max aggregator is not well suited to the embedding generation task. This is because taking the max point wise of all neighborhood vectors simply grabs the max value along all axes, which may not be representative of a node v in its neighborhood. Other aggregators like average and mean appear to be more natural, robust choices when compared to max.

8.5. Recurrent Model Performance

When comparing our two recurrent models, the RNN baseline and Graph-Sage-LSTM, we see that Graph-Sage-LSTM performs considerably better. We hypothesize that the explicit reshuffling, which trains the LSTM to work in a order-invariant fashion is effective whereas our RNN model does not handle order. Additionally, the LSTM has an additional memory cell, which faciliates what to remember and what to forget from the passed in input sequence of neighborhood vectors. The ability to select relevant information from each neighborhood vector is important, and by definition, is a much harder task for a vanilla RNN, which does not have the additional memory cell.

8.6. Random Walk Model Performance

We also see that our Random Walk model, as hypothesized, trains faster than the baseline, as seen in 4, and converges on a lower loss. Again, while it maintains virtually the same architecture as the baseline [3], it augments the neighborhood by adding random edges. We correctly hypothesized that this would accelerate the global graph learning process and produce richer embeddings, as we know that the baseline slowly incorporates more global information with each layer in the GCN.

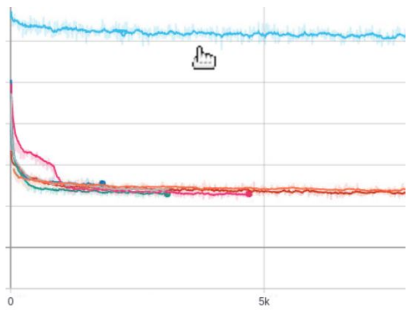


Figure 5: Training loss for various models (Zoomed-Out)

8.7. Bounding Box Loss

Bounding box prediction is the next step in the pipeline after embedding generation, so it's important to examine bounding box loss as a quick indicator of embedding utility. We note several phenomena here in Figure 6 and Figure 7: our Graph-Sage-Mean model has the lowest training and validation bounding box losses. Additionally, our Graph-Sage-LSTM, which has the 2nd best training bounding box loss is actually 4th in validation loss, which suggests that the Graph-Sage-LSTM is overfitting to the training data. We also see that our Graph-Attention-Network model also performs the worst on the validation set, which could also suggest overfitting or another issue. The learned similarity matrix for our Attention model could have been biased towards our training data. Our Random Walk model performs well on the validation set as well, matching its relative performance on the test set. This suggests that the Random Walk model generalizes well to unseen data.

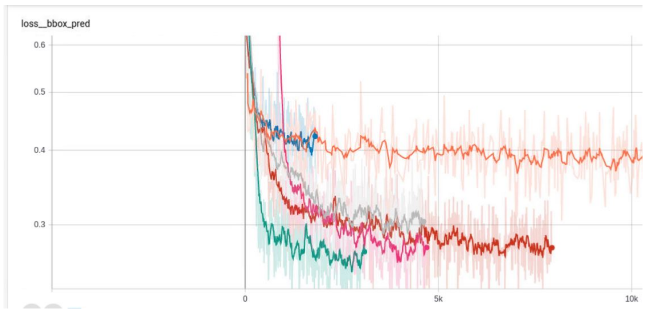


Figure 6: Training Loss for Bounding-Box Prediction

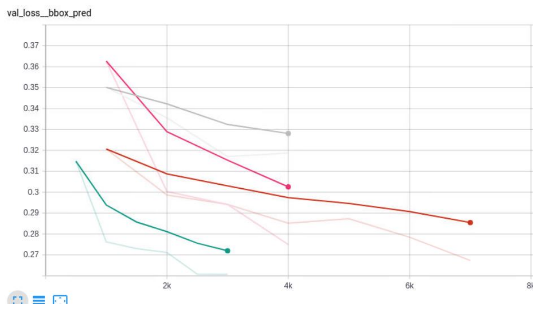


Figure 7: Validation Loss for Bounding-Box Prediction

9. Next Steps and Future Work

Now that we've established our baselines with common hyperparameters, we need to do hyperparameter search for the number of convolutional layers, encoding size, etc. As seen in many NLP tasks, increasing dimensionality boosts expressiveness of embeddings as more information can be stored. This may help in our task to create richer object and relationship embeddings. Additionally, we would like to implement ensembles of the various models where embeddings can be averaged based on the number of models in the ensemble.

References

[1] W. L. Hamilton, R. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *NIPS*, 2017.

[2] W. L. Hamilton, R. Ying, and J. Leskovec. Representation learning on graphs: Methods and applications. *CoRR*, abs/1709.05584, 2017.

[3] J. Johnson, A. Gupta, and L. Fei-Fei. Image generation from scene graphs. *CoRR*, abs/1804.01622, 2018.

[4] R. Krishna, Y. Zhu, O. Groth, J. Johnson, K. Hata, J. Kravitz, S. Chen, Y. Kalantidis, L.-J. Li, D. A. Shamma, M. Bernstein, and L. Fei-Fei. Visual genome: Connecting language and vision using crowdsourced dense image annotations. 2016.

[5] T. Lin, M. Maire, S. J. Belongie, L. D. Bourdev, R. B. Girshick, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[6] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu. A comprehensive survey on graph neural networks. *CoRR*, abs/1901.00596, 2019.