Stacked Attention Networks for Visual Question Answering

Iyer, Abhiram
abiyer@ucsd.edu

Mahadevan, Aravind
armahade@ucsd.edu

Sengupta, Neil
nesengup@ucsd.edu

Abstract

This paper presents an implementation of Stacked Attention Networks (SAN) for visual question answering task. Since Image Question answering requires multiple inference steps, the layered SAN architecture we implemented is able to progressively query images and contextually ground questions in the visual space. The architecture is then tested out on the MSCOCO dataset. Our best model is able to get a 20.7% accuracy on the dataset. We observed that the final architecture is able to relate questions to visual cues and give us reasonable answers to fairly complex questions, that is better than guessing.

1 Introduction

There has been increasing research in Computer Vision over the last few years which has led to the development of many novel methods. The goal of Computer Vision is to give computers the ability to see and understand images and scenes similar to humans. There has also been a lot of research in Machine Vision and Perception. One problem in this field includes VQA - Visual Question Answering. The goal is to teach the computer to answer natural language questions about any image. A VQA system [1] can be defined as a system that takes an image, a natural language question about the image and then is able to provide a natural language answer to the question that is asked. This is different from object detection and recognition in the sense that in object detection, the idea is to fit a given set of labels to a visual space, whereas in VQA, we need localized and contextual grounding of phrases in visual space. Thus, an important task while implementing a VQA system is to visually ground image features, natural language features in order to identify subjects and objects in a particular image. Search has to be performed over the content of the image. For example, if we want to answer a question "Are there children in this image?", the model needs world knowledge of what children are for example – visually ground natural language phrases. This problem has several applications for instance - security systems, aiding blind people (Facebook uses VQA for this) [2], surveillance etc. In this paper we build a VQA system using stacked attention networks that consists of an image model, question model and attention layers. We start off by discussing our architecture, detail our training process, analyze our different approaches finally followed by our results and a sample visualization. The challenges while implementing this includes avoiding bias, making sure we are capturing the right features for each model, getting variance in image question pairs - in order to be able to do successful visual grounding. **Figure 1** shows a sample question-answer(s) scenario that a VQA system can produce.

2 Stacked Attention Networks (SANs)

The stacked attention network consists of three sub-modules: the image model, question model, and the attention network. The overall architecture is shown below and each sub-modules will be discussed in more detail below.

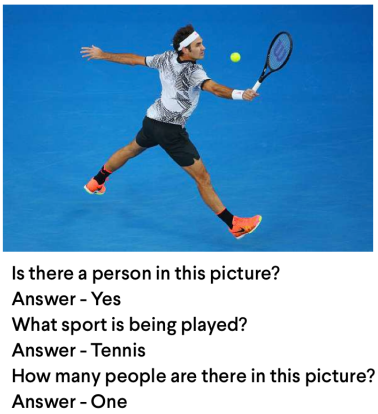

Figure 1: Sample VQA output image, question-answer

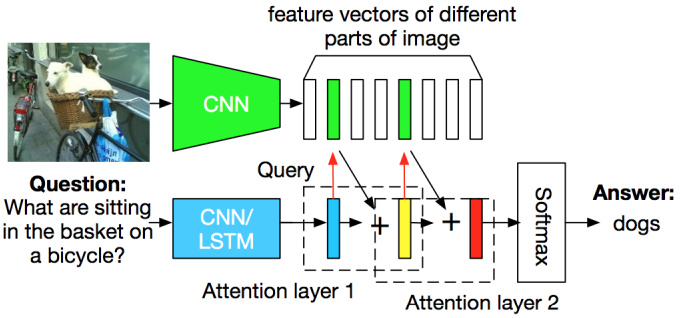

Figure 2: Stacked Attention Network architectures

2.1 Image Model

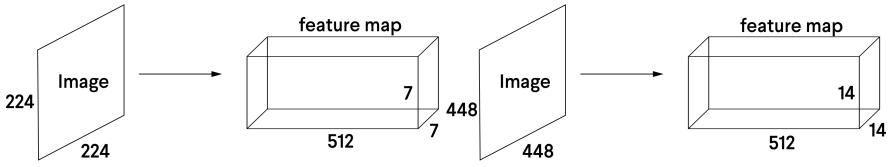

Figure 3: Image to feature map transformation using VGGNet-16 when using different image input size

The image model uses a convolutional neural network (CNN) to extract features from the image. Following the image model description in [4], we use VGGNet-16 to extract features from the image. Similar to the experimental setup described by Yang et al. in [4], the feature map extracted are from the last pooling layer from VGGNet-16. The feature maps will have different dimensions based on the dimensions of the image image as seen in 3. Using an input image with size 448×448, we obtain a feature map with dimension $512 \times 14 \times 14$. We also explored images with size 224×224 which yields feature maps with dimension $512 \times 7 \times 7$. The input pre-processing is talked more in detail below in section 3. Once the feature maps are extracted from VGGNet-16 for images of size 448×448, we transform the feature maps by vectorizing each of the 14×14 feature map into a matrix $f_I \in \mathbb{R}^{512 \times 196}$. We follow the same steps regardless of the input image size. If input image size is 224×224, $f_I \in \mathbb{R}^{512 \times 49}$. Note, we use a pre-trained VGGNet-16 which means we freeze these layers to avoid parameters getting updated during training. We finally then use a fully connected layer with a hyperbolic tan activation function to transform the feature map so that it has the same dimensions as the question vector. The equation for the fully connected layer is:

$v_I = tanh(W_I f_I + b_I)$, where v_I is a matrix such that column-i corresponds to the feature vector for region-i. Unlike the experimental setup described in [4], we also initialized the weights of the fully connected layer using Xavier initialization. The output $v_I \in \mathbb{R}^{d \times m}$ where d is the image representation dimension which is 1024 regardless of the image size and m is 196 if the original image size is 448×448 and 49 if the original image size is 224×224 number of regions which varies with input image size. In this model, the only parameters that get updated are the weights and biases of the fully connected layer described.

2.2 Question Model

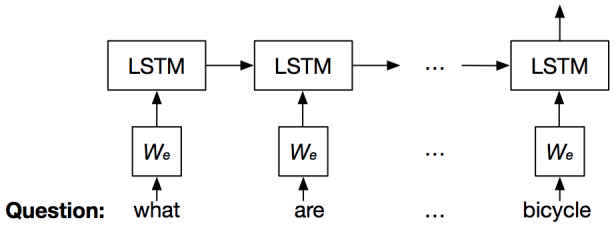

Figure 4: Question Model Architecture

For the question model, we use an LSTM to capture the semantic meaning of the question that is passed into this model. Before passing the question to the LSTM, there are some preliminary steps that need to be done. First, the questions are one hot encoded. The one hot encoding strategies are discusses more in detail in the next section as we experimented with different encoding strategies. Then given a question $q = [q_1, q_2, ...q_S]$ where q_i is the one hot encoded word at position i and S corresponds to the maximum question length which we find during the preprocessing steps described in the next section, the question is then embedded to a lower dimension using an embedding matrix. At this step we obtained a embedded question vector $x = W_e q$ where x is the embedded question vector and W_e is the embedding matrix. We also apply a masking on the embedded question vector before passing it as input to the LSTM. We mask the embedded padding word vectors in the embedded question vector so that the LSTM only considers embedded non-padding word vector. At this step, we obtain a masked embedded question vector x' which is then passed into the LSTM. We obtain the $h_t = LSTM(x')$ where $v_q = h_t \in \mathbb{R}^{1024}$ corresponds to the question representation vector from the final hidden layer of the LSTM. Furthermore, in this model, we use Xavier initialization on all learnable parameters which are the weights and biases of the embedding matrix and LSTM.

2.3 Attention Network

The attention network takes in the image representation vector, v_i, that is outputted from the image model and the question representation vector, v_q, that is outputted from the question model to predict the final answer. First we use the image representation vector and question representation vector to evaluate the following equation:

$$\mathrm{h}_A = tanh(W_{I,A} v_i \oplus (W_{Q,A} v_q + b_A)$$
$$p_I = softmax(W_P h_A + b_P)$$

To evaluate this equation, we use two single layer fully connected neural networks. We use a single layer neural network with no bias to evaluate $W_{I,A} v_i$. $W_{I,A} \in \mathbb{R}^{k \times d}$ where d is the image representation vector dimension, 1024, and k is a hyperparameter that we chose to be 512. Another single layer fully connected neural network with bias to evaluate $W_{Q,A} v_q + b_A$ equation where $W_{Q,A} \in \mathbb{R}^{k \times d}$ and $b_A \in \mathbb{R}^k$. To perform matrix vector addition, we use broadcasting techniques

inherent to python in order to compute the matrix vector addition. After performing hyperbolic tangent activation function, we obtain h_A. We pass h_A to another fully connected layer with softmax activation function to obtain p_i which is referred to as the attention distribution in [4]. Once we obtain the attention distribution, we compute $\tilde{v_i}, u$ where $\tilde{v_i}$ is a weighted sum of the attention distribution p_i and all image regions in v_i. Once we $\tilde{v_i} = \sum p_i v_i$, we then add $\tilde{v_i}$ with v_q to get a query vector u. This process described above is repeated based on the number of attention layers that is defined. More specifically, Yang et al. in [4] does the following at every k-th attention layer where $k \geq 1$.

$$h_A^k = tanh(W_{I,A}^k v_I \oplus (W_{Q,A}^k u^{k-1} + b_A^k))$$
$$p_I^k = softmax(W_P^k h_A^k + b_P^k)$$
$$u_0 = v_q$$

We use two attention layers as [4] states that it produces the best results and going over two attention layers does not result in more improvement. Once we finish the operations at the last attention layer, the authors use a fully connected layer and softmax activation function:

$$p_{ans} = softmax(W_u u^k + b_u)$$

In our implementation, we use a fully connected layer but use a linear activation function rather than a softmax activation function due to how cross entropy loss is calculated in PyTorch. In PyTorch, cross entropy loss does a logarithmic softmax followed by computing the negative log likelihood loss [3]. Due to this fact, we avoided using a softmax activation function at the last layer of final attention layer in our implementation of the attention network. All the weights and biases of all the fully connected layers described above are initialized using Xavier initialization.

3 Experimental Setting

All experiments were conducted on the MSCOCO dataset. The dataset contained 82,783 images, where each image had an associated set of questions and images – in total, there were 443,757 questions and 4,437,570 answers. Questions are subdivided into various groups (known as "question type"), which is ultimately determined by how the question is asked: the question can be open-ended, have a numerical response, a yes or no response, etc [1]. Every question has a corresponding set of possible answers, each marked with a confidence level. For the purposes of this experiment, we pick the most frequent answer as our ground-truth answer for a given question. Additionally, we only consider a question and its corresponding image if it has a one-word answer and an answer that occurs in the top 1000 most frequently occurring answers. After these initial pre-processing steps, the dataset is distilled to about 380,000 questions and answers, each with a corresponding image.

Each image is processed before being input to the image model. Images are all normalized in the range [-1, 1]. The image was either inputted as is (with original size 448×448) or center-cropped to a size of 224×224 before being input to the image model.

Questions and answers are also pre-processed before being input to the question model. Punctuation, capitalization, and special characters are removed from each phrase, followed by a one-hot encoding to vectorize each question and answer.

We experimented with various one-hot encoding strategies for the question. Initially, each question is one-hot encoded as a NxS matrix, where N is the size of the vocabulary (based on all the questions in the data, where N is about 12,000) and S is the sequence length of the longest question (22 elements). The matrix was originally constructed to be fed into a linear layer as a means to learn word embeddings onto a lower dimensional space (dimensionality reduction from N to 1000), but lack of GPU memory forced us to adapt our matrix into a single 1xS vector. Each element in the vector is simply a unique index describing the word (the rest of the elements are zero-filled and simply act as a padding before being input to the word embedding and LSTM). The answer is one-encoded in a similar fashion – each answer is associated with a unique index to describe the word.

The model was trained with cross entropy loss, where both log softmax and negative log-likelihood were applied on the output : $H_{y'}(y) := -\sum_i y'_i \log(y_i)$.[3] The index corresponding with the maximal element in the final output layer was checked against the ground truth answer index to determine the accuracy of our model. In our experiments, the model's predicted guess needed to be the exact same as the ground truth answer in order to be considered correct.

The first experiment (without normalization on the LSTM) ran with a batch size of 40, with images of original size (448x448). The whole model only used about 31.4% of the overall dataset to train and evaluate due to memory constraints. Of this 31.4%, 80% of the dataset was used to train and 20% was used as a validation set. An Adam optimizer was used with learning rate 0.01 and weight decay 0.5.

The second and third experiments (still without normalization on the LSTM) ran with a batch size of 150 and 250 respectively. For all experiments including and following these two, images were scaled down to a size of 224x224. The same training and validation splits were used as before, along with the same optimizer and hyperparameters.

The final experiment ran with normalization on the LSTM, with a batch size of 200. This experiment ran with a RMSprop optimizer, with a learning rate of 0.0004, alpha value of 0.99, epsilon value of 1E-8 and momentum of 0.9. This experiment was trained on 50% of the dataset. 80% of this reduced set was used to train the model, and another 20% was used for validation.

Furthermore, all these experiments we ran on an NVIDIA 1080-Ti GPU and training time of 10 epochs took 12 hours with 12GB GRAM.

4 Results

With the outlined experiments detailed above, below we present the results from all our experiments below. Figures 5,6,7,8,9 and 10 represent the loss and accuracy over 10 epochs for three different experiments. Figures 11 and 12 show a simple test case for our final VQA system. Table 1 shows our final training and validation accuracies after 10 epochs for the three experiments we conducted.

Table 1: Training and Validation accuracies.

Batch Size	**Training Accuracy %**	**Validation Accuracy %**
150	32.88	32.77
200	20.70	20.48
250	54.74	53.76

Figure 5: Loss for batch size 150

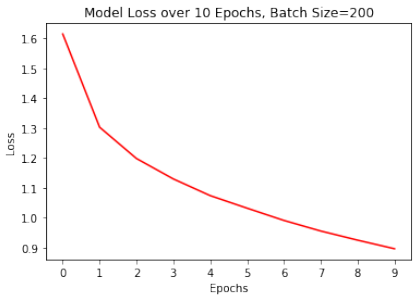

Figure 6: Loss for batch size 200

Figure 7: Loss for batch size 250

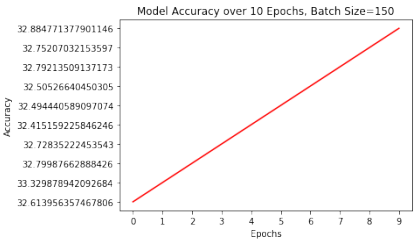

Figure 8: Accuracy for batch size 150

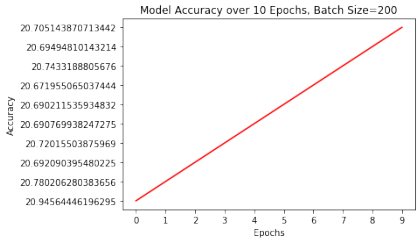

Figure 9: Accuracy for batch size 200

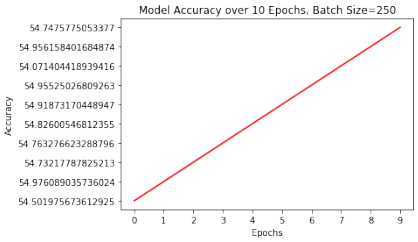

Figure 10: Accuracy for batch size 250

```
Question: has a bite been taken --> Answer: beef
```

Figure 11: VQA answering incorrectly

```
Question: are the dogs thirsty --> Answer: very
```

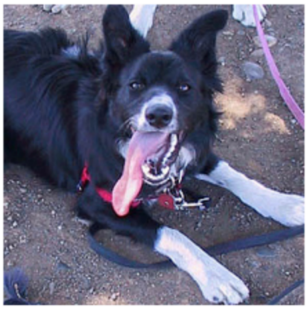

Figure 12: VQA answering correctly

After running these experiments for 10 epochs, we can clearly see that the loss decreases dramatically at the end of the first epoch and changes nominally. Although the validation and training accuracy for batch size of 150 and 250 were high, these were the cases where the system only outputted the "yes" or "no" as answers. Furthermore when training with batch size of 150 and 250, the LSTM cells were not properly initialized. The training accuracy and the validation accuracy across different experiments are very similar due to the fact that the image, question, answer triplet came from the same data distribution which explains why the accuracy are similar. Also when looking at training accuracy curves above in Figure 8, Figure 9, and Figure 10, the accuracy only nominally changes and fluctuates at around the same value. Furthermore, we did not decide to increase the epoch based on the behaviors of the loss curves seen above in Figure 5, Figure 6, and Figure 7. This indicates that the loss manifold contains many local minimas. After epoch 1, it looks like the parameters are not updating much due to the fact that the gradient may be getting stuck at a local minima. Furthermore, although the final model which was trained on a batch size of 200 does not achieve a high accuracy, it still is able to learn to answer different questions as seen in Figure 11 and Figure 12.

5 Discussion and Future Work

Along the way, we went through several iterations of our model. At the very beginning, we chose to use a linear layer instead of a dedicated embedding module to perform the word embeddings onto a lower dimensional space. As an input to the linear layer, we used a $N \times S$ matrix (where N was the size of our vocabulary and S was the length of the longest question sequence). However, the linear layer was unable to distinguish between the one-hot encoded question and the padded zero columns following it, creating problems like exploding loss.

To rectify this problem, we decided to use a dedicated word embedding module (which fundamentally acts like a look-up table for each question that maps one-hot encoded question indices from an N-dimensional space to a 1000-dimensional space). However, this did not completely solve the problem. The one-hot-encode that we used for padding and the one-hot-encode that we used for another word was the same. We rectified this issue and we applied a mask so that the LSTM stopped reading inputs from the embedded question once it reached the padding character.

We also noticed that during training time, the loss values during each minibatch never changed. To solve this problem and create a fluctuating loss value, we decided to uniformly initialize (Xavier initialize) each layer used in the model. We used Xavier initialization over other initializations such as He due to the fact that we never used relu activation and the activation functions we used were either hyperbolic tangent or linear.

After these initial fixes, we ran into our biggest problem: overfitting. At first, we included code to stop the training once the validation accuracy started to decrease (early-stopping). However, we soon realized that the validation accuracy continued to increase while the training accuracy increased – upon deeper examination, we noticed that the net was predicting the same answer for every question and image, and that this answer was always "no" (about 82,000 of the total answers were "no"). Naively, we removed all "yes" and "no" answers from the dataset before beginning training again (since these answers accounted for about 160,000 of the total), but this did not solve the problem – the net instead learned and outputted the next most frequently occurring answer. In essence, the net only learned the average answer in the answer distribution.

Again, after closer inspection, we realized that we did not initialize our LSTM parameters properly. After initializing all the cells in the LSTM (done in batches of four so that each initialized group would be different from the others), we found that net was no longer overfitting – different image and question pairs yielded different answers. Early-stopping strategies were applied again, but validation accuracies nominally changed through epochs (due to a converge in loss and training accuracy). The final model was able to successfully answer about 20% of questions in the validation and training sets.

In the future, we can input the images into the question model in their original size (448x448), as we suspect that down-scaling the images to a size of 224x224 resulted in a loss of features. Originally, down-scaling was implemented to allow larger batch sizes (memory constraints prevented larger batch sizes with 448x448 sized images), but these experiments can carried out on a GPU with more GRAM to allow for larger batch sizes with the originally sized images, thereby potentially leading to higher accuracy. The current model also uses a VGGNet-16 net in order to extract image features, as well as a shallow LSTM for the question model. In future experiments, we can also try other CNN architectures (like ResNet) and deeper LSTM architectures and examine the changes in loss and accuracy. Also to avoid memory issues, we believe that we can cache the image features obtained by the VGGNet-16 and then pass it to the the image model. This will allows us to train with a bigger batch size as we have effectively reduced the number of overall pixels that needs to be transferred to GPU.

6 Conclusion

In this paper, we describe the implementation of stacked attention network for image question answering and challenges we faced. For this network, we use two attention layers, an LSTM for the question model, a VGGNet-16 to extract features from the images and single fully connected layer for the image model. We were able to successfully train our model with a batch size of 200 and demonstrated that it is able to answer questions for a given image.

7 Appendix

The link to our project is https://github.com/abhi-iyer/visual-question-answering.

References

[1] Stanislaw Antol, Aishwarya Agrawal, Jiasen Lu, Margaret Mitchell, Dhruv Batra, C. Lawrence Zitnick, and Devi Parikh. VQA: visual question answering. *CoRR*, abs/1505.00468, 2015.

[2] Yu Jiang, Vivek Natarajan, Xinlei Chen, Marcus Rohrbach, Dhruv Batra, and Devi Parikh. Pythia v0.1: the winning entry to the VQA challenge 2018. *CoRR*, abs/1807.09956, 2018.

[3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. In *NIPS-W*, 2017.

[4] Zichao Yang, Xiaodong He, Jianfeng Gao, Li Deng, and Alexander J. Smola. Stacked attention networks for image question answering. *CoRR*, abs/1511.02274, 2015.