Team 27: Final-Term Project Report

G Karthik Balaji
CS19BTECH11001

Rachit Keerti Das
CS19BTECH11034

Gunangad Pal Singh Narula
CS19BTECH11035

Krishn Vishwas Kher
ES19BTECH11015

M. Bharat Chandra
ES19BTECH11016

Abstract

Our goal is to develop deep learning models that understand the semantic relationship between text captions and images. Our initial aim is to develop models that can produce accurate captions for images. In other words, the model learns to generate sentence vectors that semantically capture the visual information in images. We would then like to go a step further and work in the other direction: the generation of images from text. Here, we would like to work with generative models that produce accurate visual representations of the provided text sentences.

1. Introduction

Deep Learning is a rapidly growing field that sees new innovations with every passing year. In particular, it has seen a wide variety of applications in Computer Vision and Natural Language Processing. The intersection of these fields has been an inquisitive topic and can possibly provide great insights into the underlying representation of deep learning models.

Image Captioning is one such example of the above discussed topic. This is an interesting topic that contributes to numerous applications such as providing accurate scenario description to the visually impaired, augmenting the functionalities of virtual assistants,etc.

The problem of image captioning refers to generating a one-line accurate description when provided an input image. Meanwhile, the problem of text to image refers to generating a ”template” image, given a brief textual description of the scene.

Our aim, through this project is to experiment with different architectures and explore the relationship between text and image information, to produce favourable results in image captioning and image generation from text.

2. Literature Review

2.1. Image To Text

There are primarily 3 approaches to image captioning:

1. Template-based captioning.
2. Retrieval-based captioning.
3. Novel caption generation.

All of these have a common base of learning different features of the given image, the relationships between different objects in the image, and the actions the objects are performing. The difference between these methods, however, comes in how they perform this step as well as the later step of mapping the learnt image features to text.

The former two are mostly traditional machine learning based approaches. In traditional machine learning, approaches to this problem involve coming up with a appropriate feature map (usually hand-crafted) using techniques such as LBP, SIFTs etc.This feature map is then passed to a good classifier (like SVMs) to learn objects/features in the image.

In template-based captioning, [12] one has a fixed set of grammar rules that partition sentences into conventional grammatical components such as nouns, verbs, prepositions, etc. and then these components are mapped to their visual counterparts which help in predicting a caption for the input image and evaluating the accuracy to which it explains the image. [2] While the captions produced are always grammatically correct, given the way this whole method is designed, they do not produce variable length captions.

In retrieval-based captioning, we have a database of available captions and our task is to find the caption from the database that describes the input image the best. Apart from that, the basic structure of this method is similar to the previous one, i.e. there is a mapping between visual attributes to language elements, a certain measure of ”closeness” is defined between an image and available captions and the sentences themselves are parsed using a standard

method. While it produces general and syntactically correct captions, it cannot produce image specific and semantically correct captions. [7, 3]

The common thing with both the above approaches is that they view the problem as a ranking task rather than a generation task. We wish to view the problem as a generation task, since we expect more accurate results using that approach.

In novel caption generation, the objective is to generate captions from a combination of the image features and a language model instead of matching to an existing captions. Most novel caption generation methods are based on Deep Learning.

With respect to the task of image captioning, deep learning models have focused on an encoder-decoder architecture, where a model is initially used to extract feature vectors from the provided image, and a different model is then used to generate a caption provided this feature vector. This task falls under the category of supervised learning, as the model is provided with *(image, caption)* pairs to guide its training.

In contemporary architectures, a deep convolutional network is used to extract the features from the images i.e. as the image encoder. While a custom convolutional architecture can be defined, usually, this is done using a pre-trained model (such as ResNet, AlexNet, VGGNet or GoogLeNet) using transfer learning. Once the features have been extracted, Recurrent Neural Networks (RNNs) have been the popular choice to generate the required captions. This is because RNNs are well-equipped to generate variable-sized outputs, which makes them ideal for the purpose of image captioning.

Recurrent Neural Networks, however, suffer from the vanishing gradients problem, where the gradient decays exponentially with time, which makes it hard to infer a global context. Long Short-Term Memory models [6](LSTMs, Hochreiter and Schmidhuber 1997) have been the preferred replacement, and in most modern architectures, LSTMs are used for the purpose of generating the image captions.

2.2. Text To Image

Generative models are a natural choice and currently the go-to for the purpose of generating images from text captions. In particular, Generative Adversarial Networks (GANs) have been very popular.

Introduced in a 2014 paper by Ian Goodfellow [4], GANs are, as the name suggests, generative models that function on the basis of adversarial frameworks, and work in an unsupervised manner. GANs have two models: the generator and the discriminator, that play a two player minimax game attempting to counter the other player. The generator is provided a noise vector as input, using which it generates the required image. On the other hand, the discriminator attempts to classify images as real and fake (where real refers to the images from the unlabeled dataset, while fake refers to the generated images). The goal of the generator is to consistently generate images that can fool the discriminator, while the discriminator's goal is to correctly distinguish the generated images from the original ones. This goal is captured by the following minimax equation.

$$\min_G \max_D F(G,D) = E_{x\sim p_{data}(x)}[log(D(x))] + E_{z\sim p_z(z)}[1 - log(D(G(z))]$$

GANs have seen a high rise in popularity since they are efficiently able to create realistic images from a given text description. An early paper [10] (by Reed et al., 2016) on this topic learns a correspondence function between text and images [9]; traditionally, a deep convolutional net is used for the image encoder, and a convolutional recurrent net (CNN-RNN) or a Long Short-Term Memory (LSTM) model is used for the text encoder. The learned text encoder is used to encode the text into a common semantic vector, which is then concatenated with a random noise vector before it is finally provided as input to the generator. Now, like traditional GANs, the generator attempts to generate images that can fool the discriminator, while the discriminator correctly attempts to classify an image as real or fake, given the image and text embedding pair as the input. The two are simultaneously trained while they try to one-up the other. The authors go one step further and even train the discriminator by using real images with fake captions, to make the model robust.

One of the more popular recent models is StackGAN [13]. The StackGAN process is divided into 2 stages. While the first GAN stage provides a basic structure and color to the subject, the second GAN stage takes the first's output as an input adds further details to the image and fixes any present errors.

Although quite effective in its work, it is quite selective in its range of output. For eg: StackGAN is trained to work on a niche type of image; say birds, flowers, etc. and you are allowed to describe the parameters for the given type.

These diverse methods and architectures highlight the versatility of deep learning, and the various potential applications in vision and natural language processing.

3. Preliminary Results

3.1. CNN Architecture Description

1) GoogLeNet [11]: GoogLeNet was introduced in 2014 and won the ImageNet challenge that year. The main aim of GoogLeNet is to reduce the computational costs and prevent overfitting in deep neural networks (large number of hidden layers and many nodes in each layer). This is a 22-layered convolutional network, a variation of the InceptionNet net-

work. The main component of GoogLeNet is the "Inception Block".

The Inception Block is a combination of 1x1, 3x3 and 5x5 filters. The input image is convolved parallelly by each of the filters and the output of each convolution is concatenated. This helps in capturing different spatial features of the image.

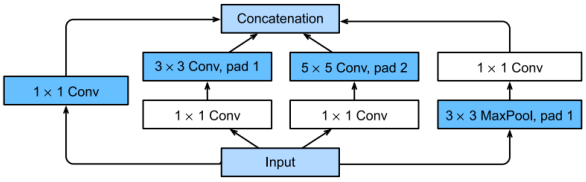

Figure 1. Working of the inception block

GoogLeNet uses a stack of 9 Inception Blocks.

Due to its sparse architecture, GoogLeNet trains faster than VGG-Net (a contemporary model), and the pretrained model occupies less space.

2) ResNet [5]: ResNets were introduced in 2015 to combat the problem of training very deep networks. Training very deep networks, can lead to unexpected results, where one achieves a higher training error than a relatively shallower network.

To help alleviate this, residual learning frameworks introduce the concept of "shortcut connections" which can be seen as skipping some layers in a feed-forward neural network. This reduces model complexity, and allows the network to remain much more computationally tractable compared to other models such as VGG-19.

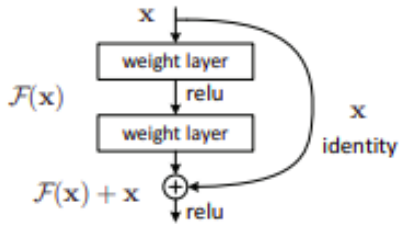

Figure 2. Building blocks of ResNets

Since the problem of training deeper networks have now been alleviated, Deep ResNets can be used to create very deep networks, and have shown notable improvement(around 28% improvement on object recognition on the COCO Dataset). A ResNet with 152 layers was presented in the ImageNet challenge, and was the deepest layer to be presented at that time. An ensemble of ResNets achieved an error of 3.57% on the ImageNet challenge, and won the challenge in 2015.

3.2. LSTM-Theory

Invented in 1997, LSTMs significantly improved upon then then existent sequence learning designs. The primary advantage that it holds in comparison to RNNs is that it alleviates the problems of vanishing and exploding gradients while also providing the provision of maintaining long term and short term dependencies. While performing backpropagation in an RNN, if one happens to encounter a contiguous sequence of gradients that are close to 0, very quickly the effect of moderately distant nodes for a particular node will go to 0 (hence the name vanishing gradients). In case of exploding gradients, the values become so large that they are infeasible to allow any further computation.

LSTMs adopt the notion of "gates" which help them get acquainted with the pattern of long term and short term dependencies that are to be preserved in the sequence. An LSTM consists of 4 gates, which are as follows:

1. *cell* gate
2. *forget* gate
3. *input* gate
4. *output* gate.

In addition, it has a "memory cell" and a hidden state (borrowed from RNNs).

In an LSTM, the memory cell helps to store vital additional information that we need to store for long term dependencies. There is an output gate which reads out entries from the memory cell. The input gates helps read entries into the cell and the forget gate helps us reset contents of the memory cells. Using the forget input gates, the LSTM can control the long and short term dependencies it wants to maintain. More specifically, the forget gate helps control the degree to which we want to retain old information while the input gate helps control the amount of additional information we want to let in.

We first mention the notation which we will be using in our equations as follows:
Number of hidden units = h,
Batch size = n,
Number of inputs = d,
Input: $\mathbf{X}_t \in \mathbb{R}^{n \times d}$,
Hidden state: $\mathbf{H}_t \in \mathbb{R}^{n \times h}$,
Input gate: $\mathbf{I}_t \in \mathbb{R}^{n \times h}$,
Forget gate: $\mathbf{F}_t \in \mathbb{R}^{n \times h}$,
Output gate: $\mathbf{O}_t \in \mathbb{R}^{n \times h}$,
Weight parameters: $\mathbf{W}_{xc}, \mathbf{W}_{xi}, \mathbf{W}_{xf}, \mathbf{W}_{xo} \in \mathbb{R}^{d \times h}$,
Bias parameters: $\mathbf{b}_c, \mathbf{b}_i, \mathbf{b}_f, \mathbf{b}_o \in \mathbb{R}^{1 \times h}$,
Candidate Memory cell: $\tilde{\mathbf{C}}_t \in \mathbb{R}^{n \times h}$,
Memory cell: $\mathbf{C}_t \in \mathbb{R}^{n \times h}$.

Mathematically the above behaviour is replicated as follows:
$\mathbf{I}_t = \sigma(\mathbf{X}_t \mathbf{W}_{xi} + \mathbf{H}_{t-1} \mathbf{W}_{hi} + \mathbf{b}_i)$ [Input gate],

$\mathbf{F}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xf} + \mathbf{H}_{t-1}\mathbf{W}_{hf} + \mathbf{b}_f)$ [Forget gate],
$\mathbf{O}_t = \sigma(\mathbf{X}_t\mathbf{W}_{xo} + \mathbf{H}_{t-1}\mathbf{W}_{ho} + \mathbf{b}_o)$[Output gate].

Next for the memory cells, we have the following mathematical equations:
$\tilde{\mathbf{C}}_t = \tanh(\mathbf{X}_t\mathbf{W}_{xc} + \mathbf{H}_{t-1}\mathbf{W}_{hc} + \mathbf{b}_c)$ [Cand. Mem-Cell]
$\mathbf{C}_t = \mathbf{F}_t \odot \mathbf{C}_{t-1} + \mathbf{I}_t \odot \tilde{\mathbf{C}}_t$ [Memory Cell].

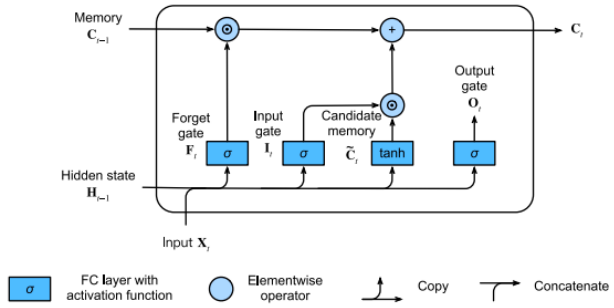

Figure 3. Computing the memory cell in an LSTM model

So if the forget cell is kept close to 1 and the input gate close to 0, the past memory cells are saved over time and passed to the current time step, thus largely alleviating the problem of vanishing gradient and helps in better maintenance of long term dependencies.

The equation for the hidden state is as follows:
$\mathbf{H}_t = \mathbf{O}_t \odot tanh(\mathbf{C}_t)$.
The tanh activation function is used primarily to restrict the range to (-1, 1) instead of (0, 1).

Whenever the output gate is approximately 1, we let the information flow and when it is close to 0, we just keep it saved within the cell and don't let it flow for any processing. [1]

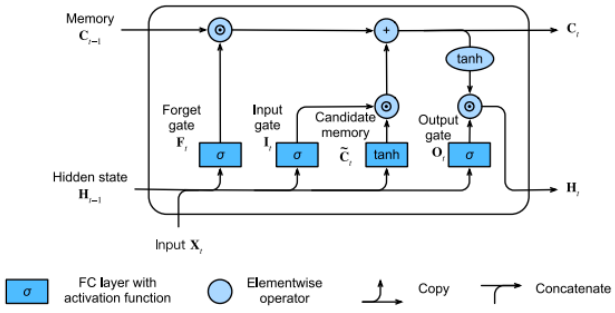

Figure 4. Computing the memory state in an LSTM model

3.3. Image Captioning: A walk-through of the code

The network consists mainly of an "encoder" and a "decoder", where the encoder is a Convolutional Neural Network (CNN) and the decoder is a Recurrent Neural Network (RNN). In this case, we use a Long Short-Term Memory (LSTM) RNN, which is a more complex version that allows the RNN to preserve relevant information over many time steps; while LSTMs provide no guarantee on the vanishing/exploding gradient problems that plague RNNs, they alleviate this issue a little and allow the network to learn long-term dependencies. The architecture is illustrated in the diagram below. [2]

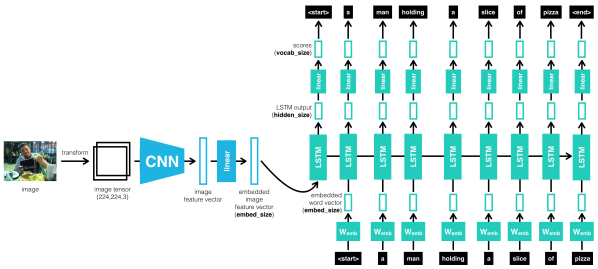

Figure 5. Encoder-Decoder Architecture

The framework used is PyTorch [8]. For the CNN, we use a pre-trained model. We experimented with both ResNet50 and GoogleNet architectures. It is generally more efficient to use these pre-trained models which are provided by PyTorch, as they have already been trained exhaustively and training a new CNN from scratch would require a lot more time. We replace the final fully connected layer of this neural net and replace it with our own linear layer, to generate an output embedding of the desired size. We then apply batch normalization on this to generate the final output embedding.

The key aspect of this encoder-decoder architecture is the embedding vector that connects them. As part of the encoder, using the linear layer, the output of the CNN is converted to a vector of a specific size, `embed_size`, and fed as the initial input to the LSTM. Similarly, using the inbuilt `PyTorch` layer `nn.Embedding`, a vector of the same size, `embed_size`, is generated as inputs to the LSTM in the subsequent time steps. This is explored in detail below.

Natural language sentences must be converted to numbers for the network to make sense of it. We first use the `nltk` library to tokenize the words i.e. split the sentence into a list of distinct words and special characters. We then generate a pair of dictionaries that defines a one-to-one mapping between the different tokens and the set of natural numbers. As a result, each image caption is associated with a unique 1D Tensor. As we can see in Figure 1, we want to feed our image embedding as an input to the LSTM on the first time step to generate the first hidden state, cell

[1] Sourced from this book

[2] Sourced from here

state and our first output token. In the subsequent steps, at each time step, we want to use the next caption token as the input, and compute the new hidden state and cell state using the previous states and current input. Since we use both the image embedding as well as the token embeddings as inputs to the LSTM, we require them to be of the same size. For this, as mentioned previously, we use `torch.nn.Embedding`, a trainable layer that allows us to generate a unique embedding vector of size `embed_size` for each token ID (since each token has a unique natural number associated with it).

Thus, we have converted the image features and the tokens in the caption to embedding vectors of a fixed size, which are then supplied as input to the LSTM. We also ensure that the captions are prepended with a `<START>` token and appended with an `<END>` token, so that the LSTM recognizes the start and end of sentences while training. We also do not supply the `<END>` token as input to the LSTM: we only want to generate it as our final output, to signal the end of sentence. Thus, the number of input sequences to the LSTM is equal to the caption length, which as a result is equal to the number of time steps in the LSTM. The interior workings of the LSTM are expressed in detail above; at each step, the LSTM generates a new hidden state vector. We once again pass this through a linear layer to generate our final score vector. This vector has as many elements as the number of elements in the vocabulary, and each index of this score vector contains the score of that particular word in the vocabulary.

During training, we fix the number of steps in each epoch beforehand. At each step, we randomly sample a caption length; this is because we want our caption length to be fixed throughout that particular mini batch, which allows us to vectorize our training. We then randomly sample a mini batch from our training set that all contain captions of the specified length, and we perform a forward and backward pass over this mini-batch. The loss function we use is Cross-Entropy loss; we want our generated output captions to identically match the captions in the training dataset, and we use the Cross-Entropy Loss function to compare our score vector with the desired output class index (in this case, it is the index of the desired token). We then perform our training for the specified number of steps and epochs over all learnable parameters (for example, we don't train our CNN parameters as they are pre-trained parameters) to obtain our final working model. In this case, it took a little under 6 hours to train the model with a pre-trained ResNet50 used as the CNN.

Thus, the overall network can be summarized as follows:

Encoder

- Pre-trained CNN, such as GoogleNet or ResNet, without the final fully connected layer
- `torch.nn.Linear`: An embedding layer that takes in the final extracted feature vector of the CNN as input and returns an output of size `embed_size`
- `torch.nn.BatchNorm1d`: A batch-norm layer for 1D tensors that normalize the embedding vector before passing it to the LSTM

Decoder

- `torch.nn.Embedding`: An Embedding layer that generates a unique embedding vector for each token in the vocabulary, which are then fed as inputs to the LSTM. This layer is also trained to generate accurate embeddings.
- The LSTM that generates a new hidden state at each time step. There are as many time steps as there are inputs (i.e. number of time steps = caption length)
- `torch.nn.Linear`: A linear layer that takes the newly generated hidden state at each time step as an input, and generates a score vector as output. This score vector has as many elements as there are tokens in the vocabulary of the dataset.

The code, along with the dataset, can be found in this drive link. [3]

3.4. Results

We trained the encoder using two pre-trained networks - ResNet50 and GoogLeNet, and generated results for both. The networks are trained and tested on the MS COCO (Microsoft Common Objects in Context) dataset.

During inference, only the image is passed as input to the LSTM. At each time step, the output generated at that time step (i.e. the token predicted) is then converted to the corresponding embedding vector and used as input to the LSTM at the next time step. In other words, we use the context of our previous words to predict our next word, as we desired of our network.

We can see that we obtain pretty good results for both encoders, but ResNet seems to give slightly more accurate results. We can also see extreme cases: the network with the GoogleNet CNN classifies the "man on the boat" as a "cake on a table"; this shows that the network is still only as good as the dataset on which it has been trained. Even though these models have been trained on a large dataset such as MS COCO, which has 83k images, the models are still susceptible to errors. We can also observe that ResNet captions the same image correctly. This can perhaps be attributed to ResNet being able to extract better features from the input

[3] The source code was replicated from here

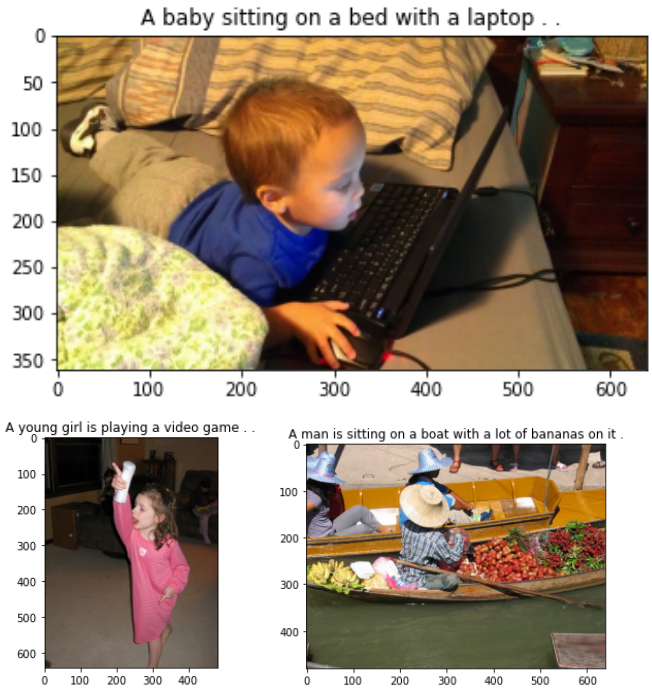

Figure 6. Results with the ResNet CNN

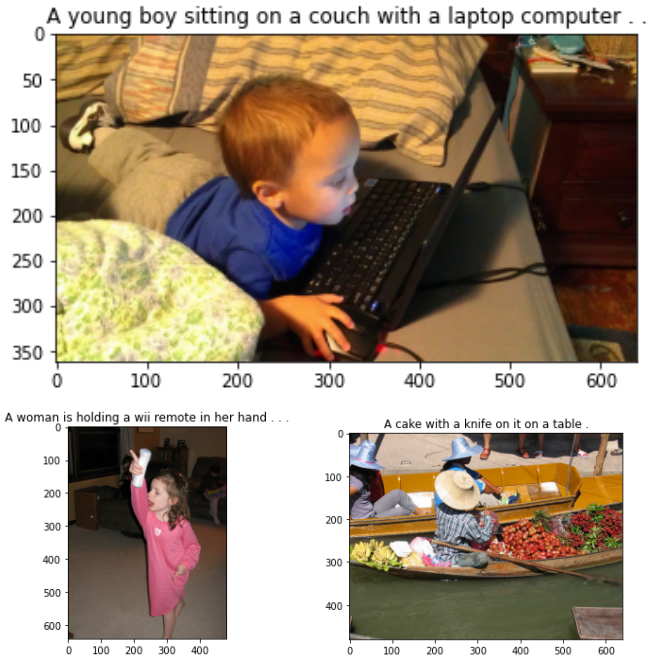

Figure 7. Results with the GoogLeNet CNN

image. This marginal improvement in performance is to be expected, as ResNet outperformed GoogleNet on the ImageNet challenge, but both CNNs perform well in general.

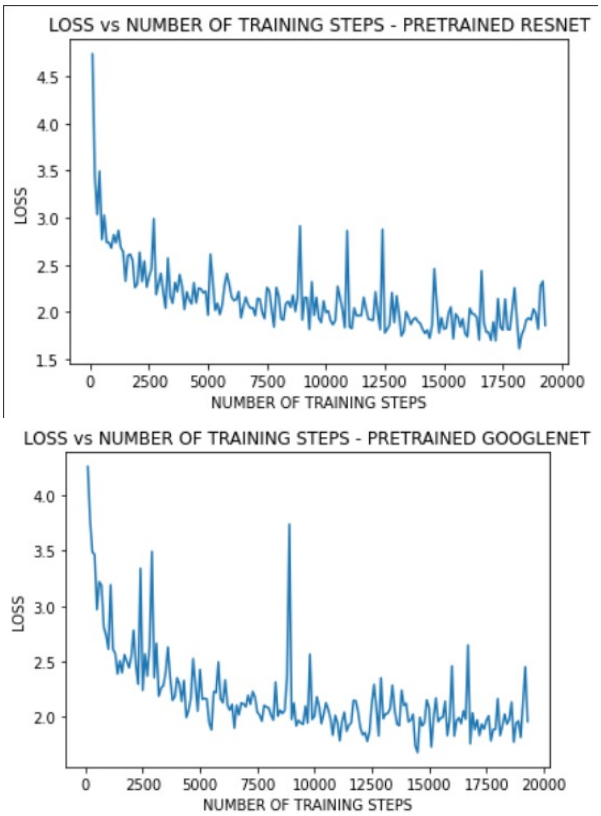

Figure 8. Loss vs Number of training steps for ResNet and GoogLeNet

4. Final Report

4.1. Text VAE Model

For modelling sequential data such as text, usually recurrent neural networks are used. RNNs usually generate outputs such as sentences, one word at a time, and don't consider a global space representation of the input.

This approach makes it tricky to use generative models such as VAEs for sentence generation. The latent spaces generated usually have "holes" present in them due to the discreteness of the text data and vocabulary.In [1], the authors present a RNN based variational auto encoder network to alleviate some of the above challenges. The proposed framework attempts to generate a global latent-space representation for a sentence. The generated latent-space representation makes it easier to capture complex details of a sentence, such as sentiment, style, topic and other high-level syntactic features.

With a VAE network, the authors aim to produce an ellipsoidial region as mapping in the latent space, rather than exact point-to-point mapping for any given input. This allows for the model to learn a smooth and interpretable feature system. As in a normal VAE, the reparametrization trick is used to model Gaussian distribution for our posterior

latent distributions $q(z|x)$, where z is a vector from the latent space, where x is the particular input. This continuous nature of the distribution allows to assign probabilities and generate new sentences by sampling from the latent space.

4.2. Our Idea and Model Architecture

We desired to combine the powers of image generation models and text generation models, and use them together to explore the intermediate latent spaces. In other words, we wanted to create a common latent representation for both images and text that would serve as a common language for these two very different representations.

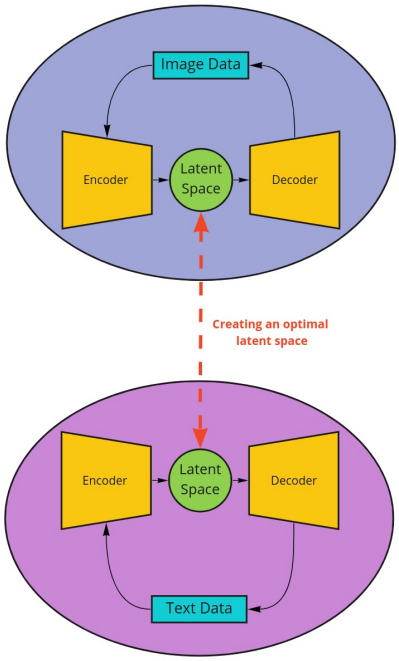

Figure 9. The model architecture

We proceeded with a novel architecture consisting of both an Image-VAE and a Text-VAE, that we simultaneously trained. The Image-VAE possessed a pre-trained ResNet50 model as its encoder, which was followed by 2 parallel linear layers that generated vectors corresponding to the mean and log-variance. Using the reparametrization trick, one could sample the latent vector from the corresponding distribution. This latent vector is then fed to a Decoder consisting of 3 Transposed Convolution Layers, that finally generate an image as output from this latent representation.

Our TextVAE is loosely based on the model described in Section 4. It also possesses a similar architecture to the ImageVAE, with the fundamental difference present in the models used for the encoder and decoder models. We use a GRU for our encoder and decoder; the encoder GRU generates a hidden state tensor, and this tensor is once again fed as input to two parallel linear layers to generate the mean and log-variance of the latent distribution. Using the reparametrization trick again, one could then sample the required latent vector. This latent representation is then fed as input to the decoder, which generates a hidden state tensor from the latent vector, which is then passed along to the decoder's GRU, which generates the required final output.

The idea behind such an architecture is that since we have a latent-space for each representation of the data (say text or image) and we train on the similarity or closeness of these latent spaces (as elaborated on below), the resulting optimal latent space we target is a universal latent space that serves as a common representation regardless of what the initial data representation was i.e. if someone wishes to train a VAE on some other data-representation/domain (say audio), they can train it assuming this "optimal latent space" as a ground truth, if such an optimal space can be achieved.

For example, in this specific instance, we wanted to explore if our trained model could eventually function as a text-to-image model, by first generating the latent representation from the text encoder of the Text-VAE, which could then be passed to the Image-VAE to generate the final image output, which would ideally correspond to the input caption.

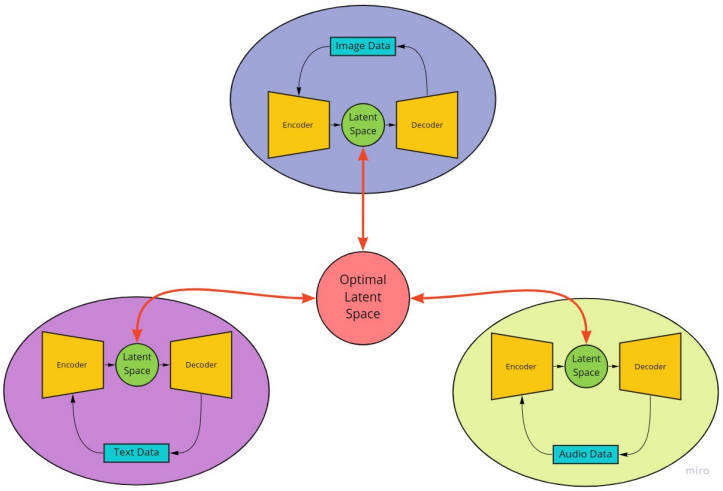

Figure 10. The future scope

4.3. Training Strategy

Since the objective was to make the two latent spaces as close to each other as possible, it required that we train both the Image-VAE and the Text-VAE simultaneously. We used the MS-COCO dataset for our purpose. Thus the overall loss consisted of 3 different terms:

- **Image-VAE Loss**: This is the loss corresponding to the individual performance of the Image-VAE. Generally, for image generation, a cross-entropy loss is used

to compare the reconstructed image with the original image. However, we encountered astronomical losses on the order of 10^{27}, and as a result, we decided to proceed with an MSE Loss for the reconstruction instead, which is simple yet sufficient for this purpose. We add the KL Divergence Loss to this loss to obtain the overall image loss.

- **Text-VAE Loss**: Similar to above, it is the loss corresponding to the individual performance of the Text-VAE. The Text-VAE's output is a tensor of dimension $caption\_length \times vocab\_size$; essentially, it generates a probability vector for $caption\_length$ different words, and each probability vector corresponds to the probabilities of all the different words in the vocabulary of the dataset. We use a cross-entropy loss in this case, as this is equivalent to training a classifier to identify the correct output class. Once again, there is also an associated KL Divergence Loss that is added to the overall loss.
- **Latent Loss**: This is probably the most important loss term amongst the 3, as it is the loss corresponding to the similarity of the two different latent spaces. Here, we once again used MSE-Loss as our metric of comparison; we decided not to directly compare the latent vectors of the Image-VAE and the Text-VAE, as the latent vectors are randomly sampled from a defined distribution, which is a random process. Instead, we compare the generated `mean` and `logvar` vectors, which are deterministic. Thus, we compute the MSE-Loss between the mean vectors of the Image-VAE and the Text-VAE, and also compute the MSE-Loss between the logvar vectors of the two VAEs, and add these terms to the overall loss.

We compute our overall loss by summing up all the individual loss terms, and backpropagating on these terms, to simultaneously train both models.

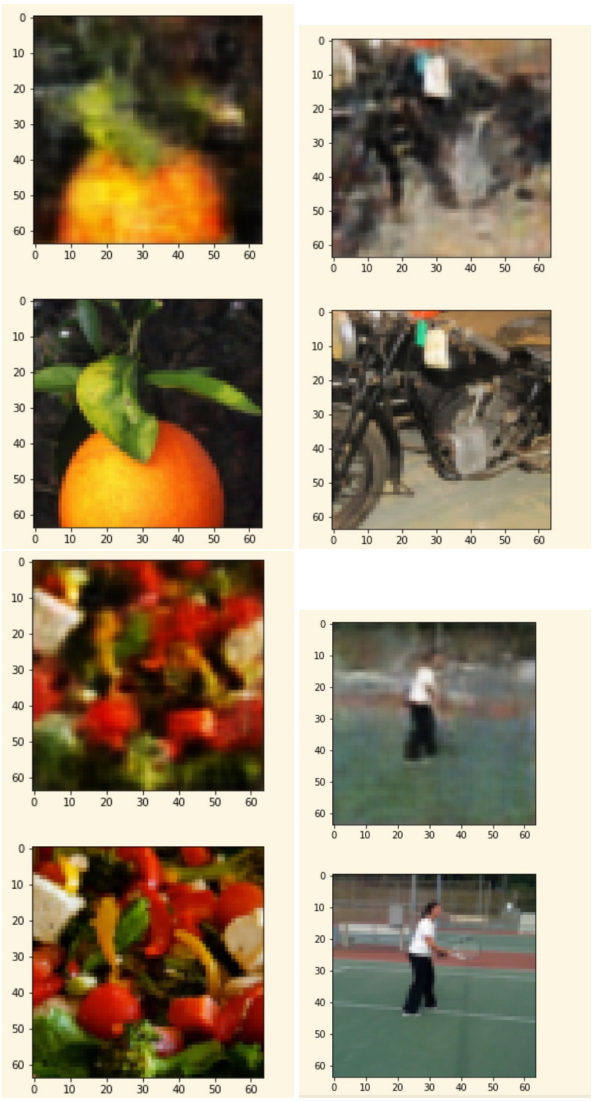

Figure 11. Some reconstructed images with the respective original images below

5. Results

We trained our model according to the training strategy described above. We experimented with the number of epochs, batch size, and tried various loss functions while training our model. Our code can be viewed through this drive link.

5.1. Image VAEs

The image VAEs were trained for around 100 epochs. Initially, the reconstructed map is a random image. After training for about 60-80 epochs, the generated image starts to greatly resemble the training image. We observed that the basic structure of the input image is more or less replicated, and only fine details are lost due to some blur effects in the generated image. Examples of some reconstructed images along with their original images are shown in Fig. 11

5.2. Text VAEs

The text VAEs also generated captions corresponding to the input captions. These could be perceived as similar to the input sentence. The generated captions had incorrect grammar at places, however it managed to capture the essential objects being described in the original caption .This was especially true when our input was a smaller subset of the entire input dataset.

The results are displayed on Fig.12

However, with a large data set, it seemed that the vocabulary of the input captions became too complex for the model to learn and produce meaningful results.

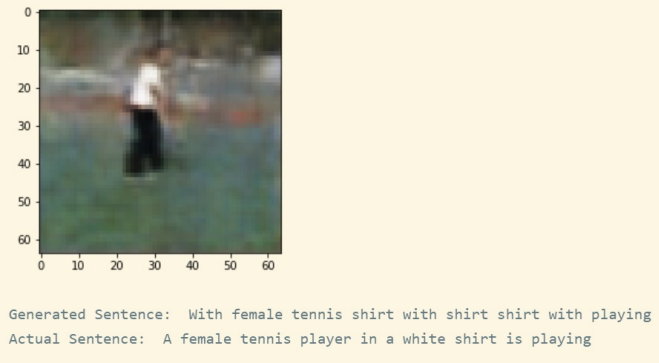

Figure 12. Generated Captions from input texts

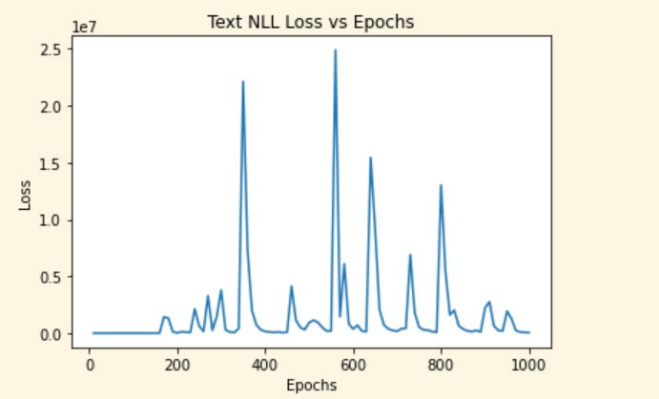

Figure 13. Text NLL Loss

5.3. Latent Spaces

We did not get satisfactory loss values on most of our epochs. The loss convergence seemed tough for the model and sharp spikes were observed in the losses. We tried different combinations, such as slightly increasing the weights of the latent-space loss, and using different loss functions. However, all of them produced similar results. This leads to the generated image not corresponding well enough to the text captions.

As seen in the given generated image (Fig.14), we do see some giraffe like texture in the image and a shape resembling its head ([0,20]x[30,55]) but the rest of the image is pretty much ambiguous.

5.4. Possible Improvements

5.4.1 Architectural Improvement to TextVAE

Currently, our TextVAE model is a very simplified version. We use GRUs for the encoder and decoder networks. LSTMs may be used to produce slightly better results for the TextVAE. A highway network is also usually suggested

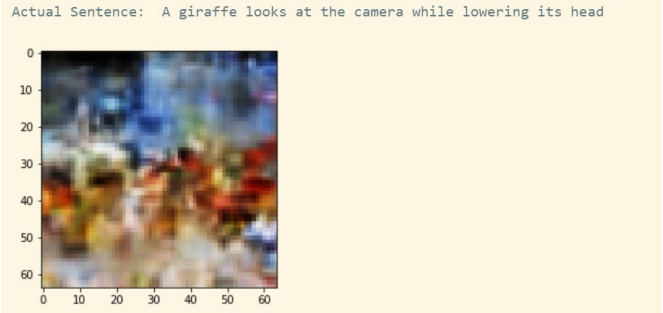

Figure 14. Generated Image for a Given Caption

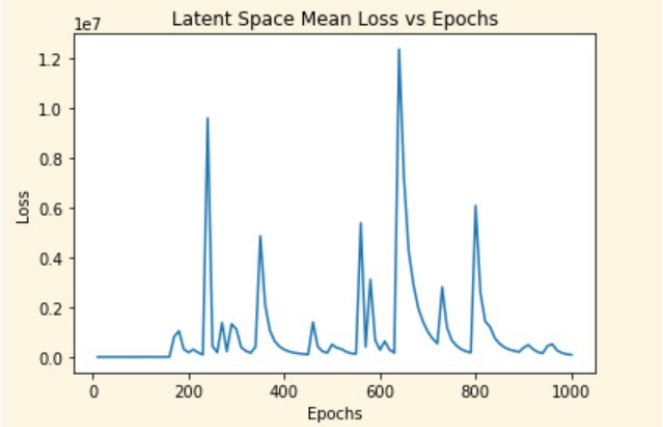

Figure 15. Latent Mean Loss

[1] to allow for smoother training and better results.

5.4.2 Loss Functions for Both Latent Spaces

Currently, we use a simple `MSELoss` for comparing the generated text and image latent spaces. Application of other sophisticated loss functions may provide somewhat better results and in speedier convergence of the two latent spaces.

It may even be possible to use a neural network to model the transformation between both the latent spaces and minimize the distance between both the latent spaces.

5.4.3 Grammar rules Enforcement

Earlier on, we discussed about template-based captioning. One idea is to use some ideas from that domain of captioning to enforce certain grammar constraints on our generated captions. One motivation for this approach of learning is the way we as humans learn to describe scenes, using sentences; and so given a certain set of words that we want to describe about a particular scene, we can use grammar rules to properly arrange the sentence. This is especially important in the current context, because in our experiments, we found that a couple of times, the generated caption contains the right set of words to describe an image, but doesn't always arrange them in the right order.12

References

[1] Samuel R. Bowman, Luke Vilnis, Oriol Vinyals, Andrew M. Dai, Rafal Jozefowicz, and Samy Bengio. Generating sentences from a continuous space. 2015.

[2] Ali Farhadi, Mohsen Hejrati, Amin Sadeghi, Peter Young, Cyrus Rashtchian, Julia Hockenmaier, and David Forsyth. Every picture tells a story: Generating sentences from images. volume 6314, pages 15–29, 09 2010.

[3] Yunchao Gong, Liwei Wang, Micah Hodosh, Julia Hockenmaier, and Svetlana Lazebnik. Improving image-sentence embeddings using large weakly annotated photo collections. In *Computer Vision, ECCV 2014 - 13th European Conference, Proceedings*, number PART 4 in Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pages 529–545, Germany, 2014. Springer. 13th European Conference on Computer Vision, ECCV 2014 ; Conference date: 06-09-2014 Through 12-09-2014.

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[6] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9:1735–80, 12 1997.

[7] Vicente Ordonez, Girish Kulkarni, and Tamara Berg. Im2text: Describing images using 1 million captioned photographs. In J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 24. Curran Associates, Inc., 2011.

[8] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[9] Scott Reed, Zeynep Akata, Honglak Lee, and Bernt Schiele. Learning deep representations of fine-grained visual descriptions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[10] Scott E. Reed, Zeynep Akata, Xinchen Yan, Lajanugen Logeswaran, Bernt Schiele, and Honglak Lee. Generative adversarial text to image synthesis. *CoRR*, abs/1605.05396, 2016.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions, 2014.

[12] Chaoyang Wang, Ziwei Zhou, and Liang Xu. An integrative review of image captioning research. *Journal of Physics: Conference Series*, 1748(4):042060, jan 2021.

[13] Han Zhang, Tao Xu, Hongsheng Li, Shaoting Zhang, Xiaogang Wang, Xiaolei Huang, and Dimitris Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. In *ICCV*, 2017.