

VISUAL QUESTION ANSWERING



INTRODUCTION

In this report, I document all the steps and procedures I used to tackle the problem of Visual Question Answering using Machine Learning. There are multiple steps to solving this problem, including handling the dataset, choosing an architecture, building + fine-tuning the model, optimising etc. Let's cover them one by one.

NOTE: All project resources can be found on the following link:

https://github.com/nikiitb/Project_2/tree/vqa_custom

Dataset



"Is the person playing a Wii?"

-Yes

Downloading + Forming the Dataset

For this project, I was supposed to use the VQA-V2 dataset (<https://visualqa.org/>). This dataset consists of a whopping 4,43,757 questions, 44,37,570 answers and 82,783 images. Each question has 10 answers, accompanied by an *answer confidence* values as well, suggesting how confident the subjects were while giving the answers. An example snippet of the above:



Question: What is this photo taken looking through?

Answer: net, netting, mesh

Now, the trick was to create a unified dataset out of this, because in it's raw state, the dataset is divided into a directory of images, and two JSON files with questions and annotations respectively. The images themselves were around 13.5GB of data.

Here were a couple of methods that I could have gone with:

1. Create an iterative dataset

- I could have used the *datasets* Python library to download and use an iterative version of the dataset. It would have saved the initial setup time and effort of creating and mapping the dataset myself, but the training time would be hugely affected because each sample would have been an element of a generator function.

2. Create the entire dataset ✓

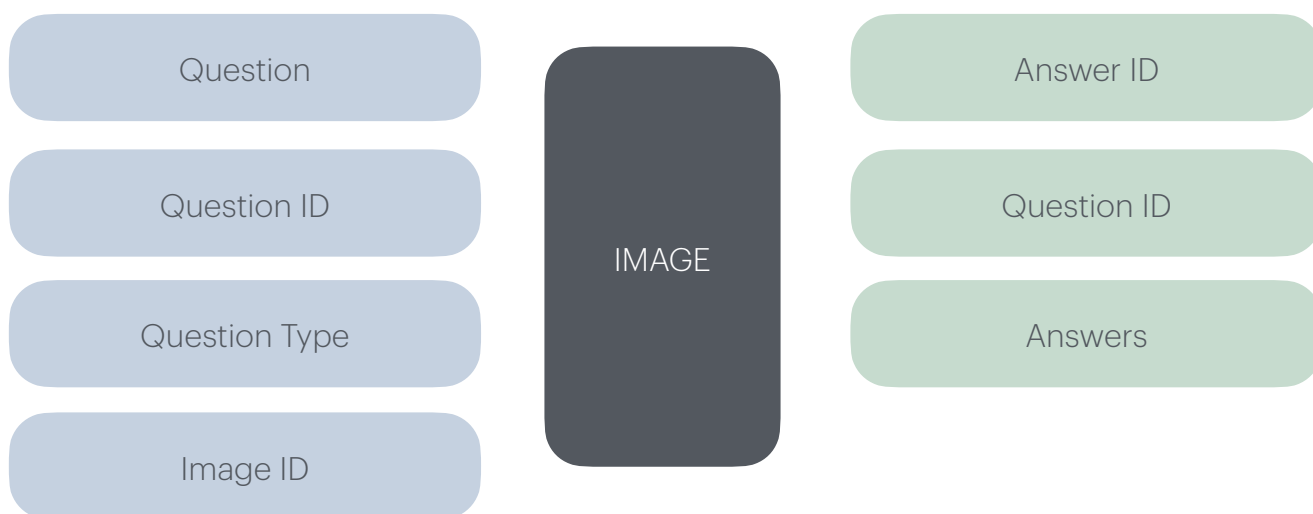
- Other way was to download the entire data, and create the dataset myself. Considering I may have to try various iterations and designs of models, I chose to go this route since this would be a one time process but would be really beneficial with multiple rounds of training loops.

So, I downloaded the entire dataset (images + json files) and used the Pandas library to create the dataset manually. The reason I went with Pandas instead of the dataset library is because Pandas is way faster for processing the dataset in one go because it is loaded into memory. On the other hand, Datasets library loads datasets from disk (Default behaviour) which makes mapping and other intermediate processing operations time consuming.

Whenever required though, I could simple choose to switch between one or the other using inbuilt APIs.

NOTE: The dataset also contains 2d images, which I manually converted to 3d before going ahead.

The Dataset



Using the Questions JSON, Answers JSON and images, I stitched them together to form one dataset of 4,43,757 rows (corresponding to 4,43,757 questions).

There was one more crucial step remaining before we could go towards designing and building a model. The answers at the moment were in a format that was pretty difficult to train. One example answer:

Question: Where is the light coming from?

Answer:

Answer	Answer Confidence
Moon	Yes
Moon	Yes
Moon	Yes
Moon	Yes
Moon	Maybe
Bulb	Yes
Bulb	Maybe
Bulb	Maybe
Street Light	Maybe

Each of the 10 answers per question is recorded by 10 subjects, and each answer has a surety metrics (sort-of) attached with it.

Labelling Scheme

We need to convert this into a trainable format. For this, I went with the following scheme:

- Assign *Answer Confidence* “yes” to a score 1, and “maybe” to zero.
- Sum up scores for each answer (among 10 answers) and normalise the score.

$$z_s = \frac{\sum_i^s x_i}{\sum_i^N x_i} \text{ where } x_i = 0.5 \text{ or } 1$$

(My labelling scheme)

Using the above scheme, for the above question, we would get a label as follows:

{“moon” : 0.64 , “bulb” : 0.29 , “street light” : 0.07}

Now we have a proper metrics attached with each label, which we can then choose what to do while training. In the VQA paper *VQA: Visual Question Answering* (<http://arxiv.org/pdf/1505.00468>) the authors have opted for a different method, where they consider any answer more than a frequency of 3 to be a sure answer, otherwise relative weightage.

Using this, we have a dataset that looks like follows:

$$z_s = \min\left(\frac{x_i}{3}, 1\right) \text{ where } x_i \text{ is count of } i\text{th answer}$$

(Labelling scheme in the original paper)

question	question_type	question_id	image_id	answer_type	label
What is this photo taken looking through?	what is this	458752000	COCO_train2014_000000458752.jpg	other	{'ids': ['net', 'netting', 'mesh'], 'weights':...
What position is this man playing?	what	458752001	COCO_train2014_000000458752.jpg	other	{'ids': ['pitcher', 'catcher'], 'weights': [0....
What color is the players shirt?	what color is the	458752002	COCO_train2014_000000458752.jpg	other	{'ids': ['orange'], 'weights': [1.0]}

Is this man a professional baseball player?	is this	458752003	COCO_train2014_000000458752.jpg	yes/ no	{'ids': ['yes', 'no'], 'weights': [0.94, 0.06]}
What color is the snow?	what color is the	262146000	COCO_train2014_000000262146.jpg	other	{'ids': ['white'], 'weights': [1.0]}

Now that I had my labelling scheme ready, I had to find a way to sample the data.

Sampling

As per the problem statement, I was allowed only to use only 1/4th the training data. Hence, out of the ~4L samples, we are allowed only ~1L samples to use. So I looked into the data, and found that there were:

1. **65 unique kinds of questions** (including “are..” , “how many..” , “is there..”) etc.
2. **3 unique kinds of answers** (yes/no, other, number)

Since the answer types were not very comprehensive, I chose to sample the dataset according to the question types. In this case, I randomly sampled equal number of questions (if available) from each types of questions.

So, sampling around 2100 questions from each of the 65 questions gave me a dataset of ~1.4L. This was a start, **but this sampling method would change a lot with experimentations in the coming sections of the paper.**

Now moving on to how to make these labels fit for feeding to an ML model. I used a simple Python dictionary to map every single answer into a unique label. This way I had a dictionary of all possible answers each with a unique label assigned. Each label had a soft label assigned to it (as calculated earlier, i.e the weighted averages of each answer). So considering the question and answer in the above example, my answer would be like a (soft) one-hot encoded vector of length equal to size of all unique answers in the dataset:

...	0	0.64	0.29	0.07	0	...
...	tiger	moon	bulb	street light	car	...

Now that the data is ready, let's move on to model design.

MODEL

Now that the dataset was completely prepared, I had to think of a way to come up with the model. Here are a few methods that I could think of:

1. Sequence to Sequence problem

- Given image and text features, the model could take them as inputs and generate the answer as another sequence of texts. This would require some sort of decoder architecture, such as a GPT. But considering that GPT was not allowed for this problem (only ViT and BERT family models were allowed) hence this solution was pretty much out of scope.

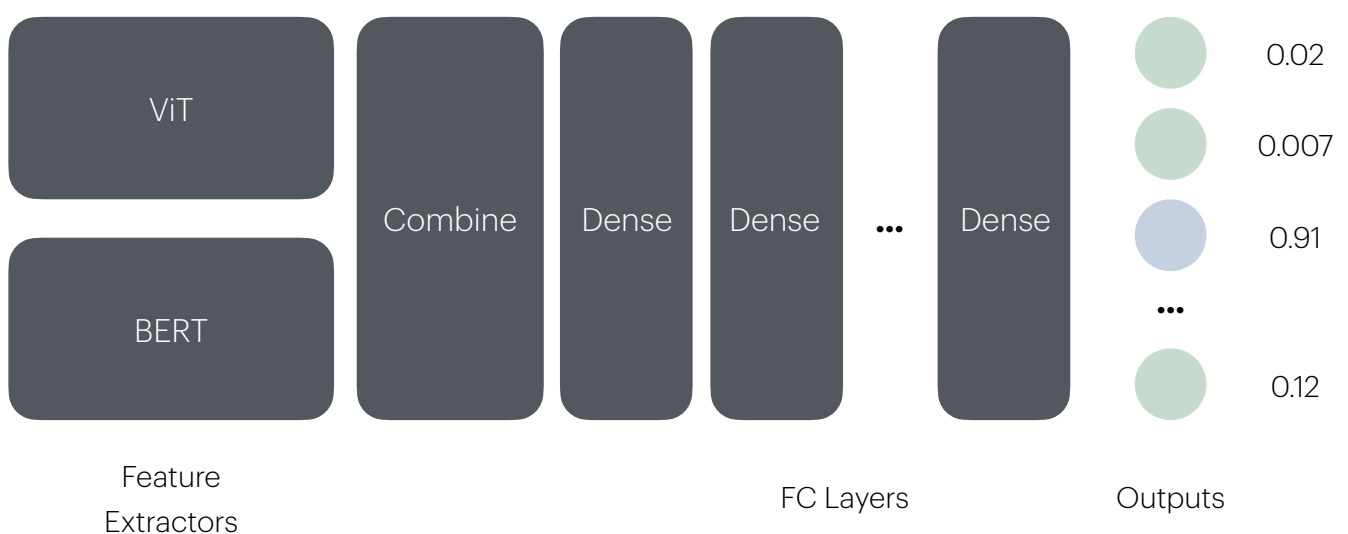
2. Classification Problem

- Treat each of the answers as a word in a vocabulary space, and similar to how decoders work, output the answer as one of these tokens in this dimensional space.
- So for example, let's say we only had 10 questions in the dataset. They all have a combined of 5 unique answers. So the classification problem is to output probability of the most probable answer.

The second approach seemed doable considering the dataset that we had and the constraints with the problem's statement.

What remained now is a method on how to combine the visual and textual features for the training. For this, I thought of using ViT and BERT as visual and textual feature extractors respectively, and feed them to my own custom model (FC layers) to get an output.

So that would result in a model as follows:



This architecture felt like a good start. I just had to figure out some way to combine the visual and text features. For this, I had two options again:

1. Multiplication

- Take the visual and text features, and do an element-wise multiplication. Optional: Have an FC layer per feature before the multiplication operation.

2. Concatenation

- Simply concatenate the features into one doubly sized array, and pass it to FC layers.

For this, I hypothesised that choosing the second option should be good enough because the Dense layers would be powerful enough to understand the nature of visual and textual features and would learn the weights accordingly.

I also had to confirm that the features should be normalised, else I would have to add my own normalisation layers. I confirmed they were, and so the extra layers were not necessary.

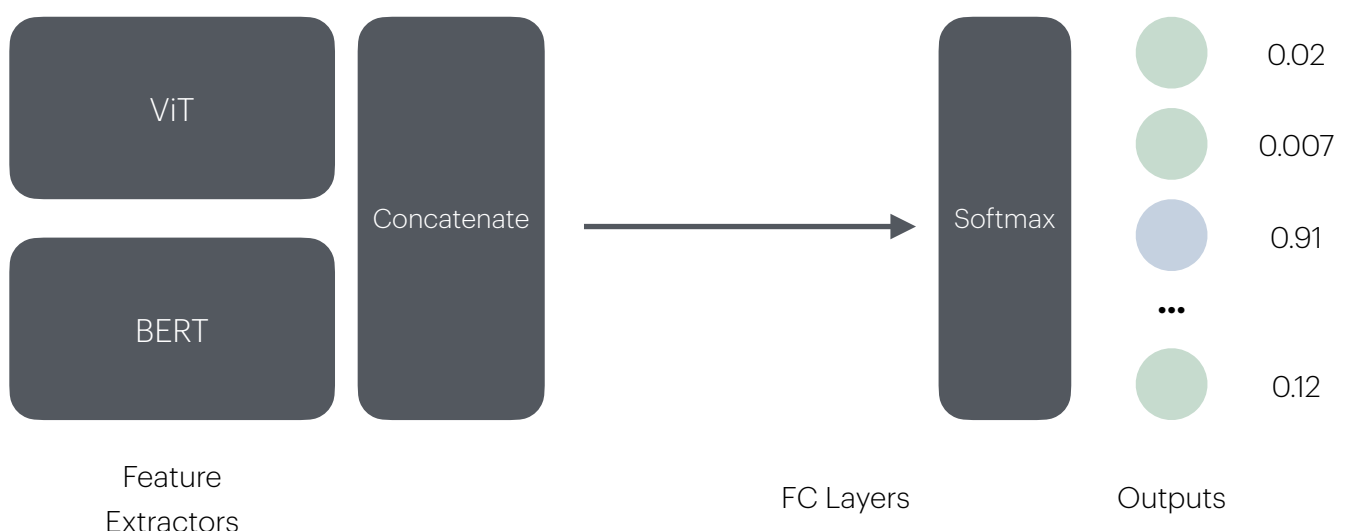
Next, I had to choose a framework for implementation, It was either **PyTorch, or TensorFlow**.

Considering I had prior experience with TensorFlow, I went ahead with it (I also use PyTorch in pages to come).

NOTE: All models have been trained on GPU: Nvidia Titan (12288 MiB) and sometimes on CPU.

I initially came up with a model as follows:

Model1

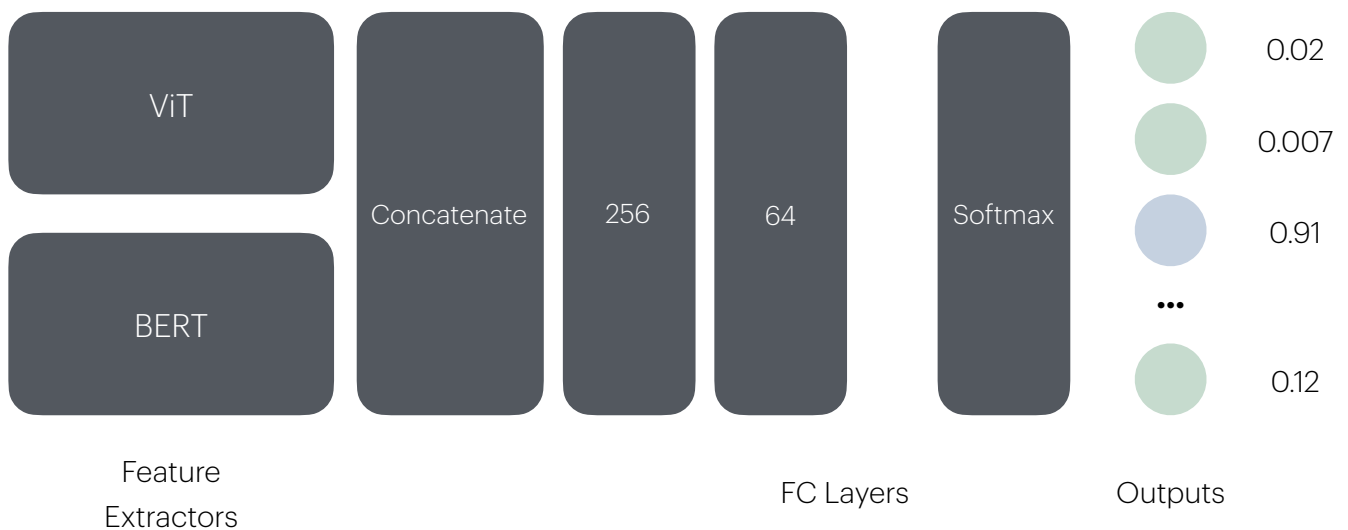


This model contains one dense layer with number of filters = output classes (softmax activation). This is the simplest possible model based on the architecture blueprint I thought of. Just for testing this architecture, I tested this on a dataset of just 100 samples.:

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	20	100
Training Time	Training Accuracy	Validation Accuracy	Testing Accuracy
46.2	0.67	0.15	0.15

Now, the validation accuracy was pretty bad, but what's more concerning is that the training accuracy was bad too. It was just 0.67. Meaning, even on a small dataset of 100 samples, the model could not converge. Hence, I modified the architecture a bit:

Model2



This model now had 2 added Dense layers, each of 256 and 64 channels respectively. Each had ReLU activation, followed by output Dense layer of Softmax activation. This model have the following output:

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	10	100
Training Time (s)	Training Accuracy	Validation Accuracy	Testing Accuracy
47.8	0.97	0.22	0.20

This model seemed perfect. It was able to overfit to the small amount of data. Validation accuracy was still low, but for that, I just needed to increase the data (or so I thought, but I was wrong as we shall see).

Before I could move on to training on higher number of samples, I faced one more issue.. the GPU was going **out of memory**. I searched for many methods to be able to solve this, the best one I found was using a **Generative Dataset**.

This way, I can use a dataset of any size because I won't be loading it into memory until the actual batch is called forth for training.

This one method helped in saving a lot of memory while training, but did not help in training time, since now, the data was being read from disk and processed dynamically while training.

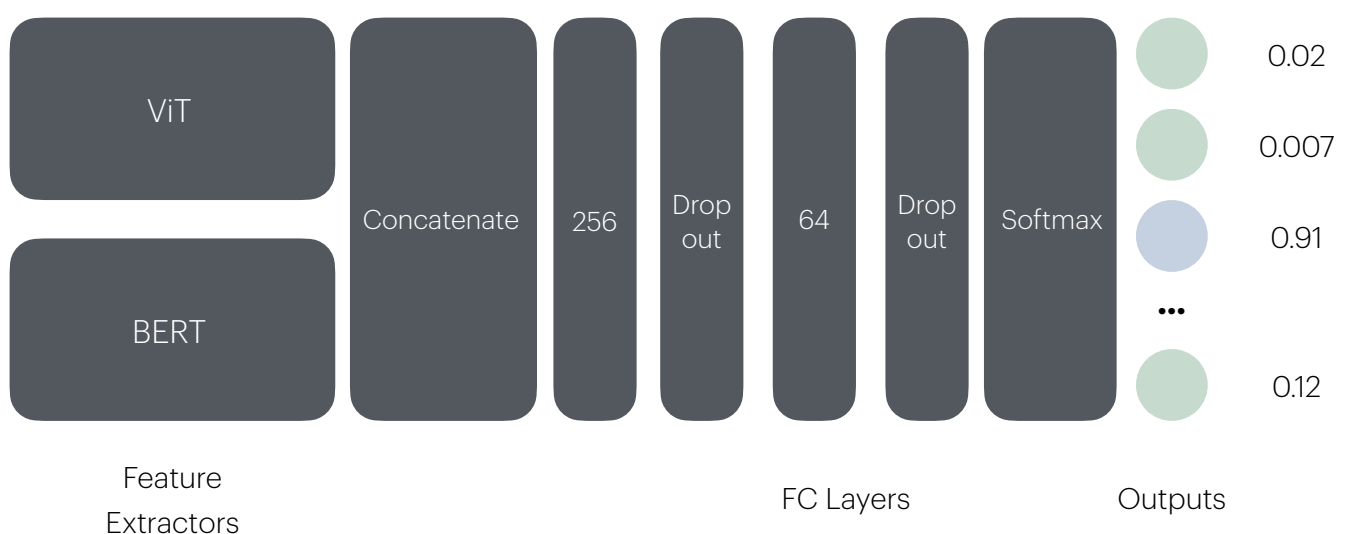
Now that the OOM error was resolved, back to training.

So I trained the above model for 30 epochs, on a dataset of 10k samples. Results:

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	30	10000
Training Time (hours)	Training Accuracy	Validation Accuracy	Testing Accuracy
1.3	0.96	0.27	0.26

Even with 10k samples, the validation accuracy was abysmal. The model was fine, since it was able to learn (evident with high training accuracy). But the validation accuracy was extremely low (in fact, it hasn't much changed since using just 100 samples). Hence, I added some regularisation to see if it helps:

Model3



This time, I added a dropout layer Infront of every Dense layer in the model. I trained again with 10k samples.

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	30	10000
Training Time (hours)	Training Accuracy	Validation Accuracy	Testing Accuracy
1.4	0.94	0.30	0.28

This helped some, but not much at all. There was a huge gap between the training and the validation accuracy. I spent a lot of time trying different hyper-parameters, different shuffling patterns etc. but to no avail. Then I stumbled upon one realisation. For the 100 samples that we had chosen earlier, the number of unique answers (and thus labels) were 179. For 10k samples, this number was 12632. Meaning, the **number of classes to be learned was too much**. In fact, it was so much that it exceeded the number of training examples. In the end, I could think one these two reasons that could be leading to bad performance:

1. Too many classes to learn

- As explained above

2. Similar inputs / lack of variance in inputs in training data

- One more reason the training data was very good but the validation accuracy was not might be because the inputs were very similar, but the validation data was completely different.
- But I checked this, and turns out this was not the case. The sampled 10000 dataset had very little repeating images, and most of them were well distributed throughout the training and validation dataset.

So then, we had to tackle the first problem, and **reduce the number of learnable classes**.

I thought of various ways this could be possible, and two major ways that I finalised were these:

1. Create a tree of models / Task specific models

- So there were three types of answers - *numbers*, *yes/no* and *others*. So in this case, we could create 3 separate models to tackle the 3 separate kinds of problems. This way, one additional input to the modal would have to be given, which is, the "output type".

2. Clip total number of learnable classes

- Out of the total number of learnable classes, we clip them to a much smaller number, and remove any questions from the datasets, answers of whom do not fall in this answer set.

Both approaches seemed like good options to try, and hence I gave them both a go.

Task-Specific Model

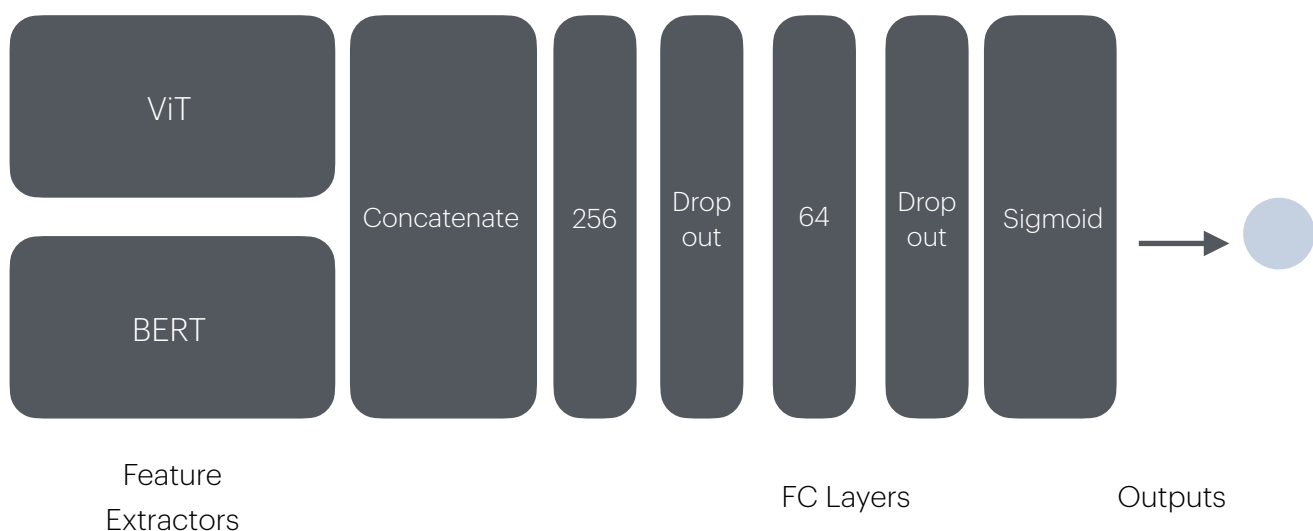
In this method, I would train models based upon specific answer types (3 models for 3 answer types - number , yes/no , other). The *other* category was similar to the original problem at hand, because those answers could be anything. So I could try either of the two models (number or yes/no) to test this method. Hence, I chose to train a model with only yes/no answers.

Sampling (modified) the Dataset

This time, I modified the sampling of the dataset by only considering answers of the type “yes/no”. From these, questions from each of 65 kinds were chosen randomly and sampled into one single dataset.

Once the modified dataset was ready, I sampled 10k of these “yes/no” questions and began training on them. I had to modify the model:

Model3a



The results:

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	20	10000
Training Time (hours)	Training Accuracy	Validation Accuracy	
1.1	0.96	0.51	

This was definitely something I was not expecting. The validation accuracy was too low for even two classes now. So I speculated something might be wrong with the data itself. So I checked the “yes/no” dataset again, and found that not all the answers were pure.

I did a label analysis, and found that even in this “yes/no” dataset, there were almost 5000 unique labels (there should have been only two, yes or no). So this means that, there was a lot of noise / impure samples in the dataset in general, and manual cleaning was required.

Cleaning the dataset

So I removed any other label from the dataset other than yes or no in the “yes/no” dataset. I performed one last sanity check to make sure everything was alright, and indeed, the total number of labels now were only 2 - yes or no.

Now that the dataset was cleaned, I did a quick train just to see everything was alright. So I sampled just 1000 datapoints, and gave them to the model:

Optimiser	Learning Rate	Epochs	Samples
Adam	3E-05	20	1000
Training Time (min)	Training Accuracy	Validation Accuracy	
7	0.99	0.77	

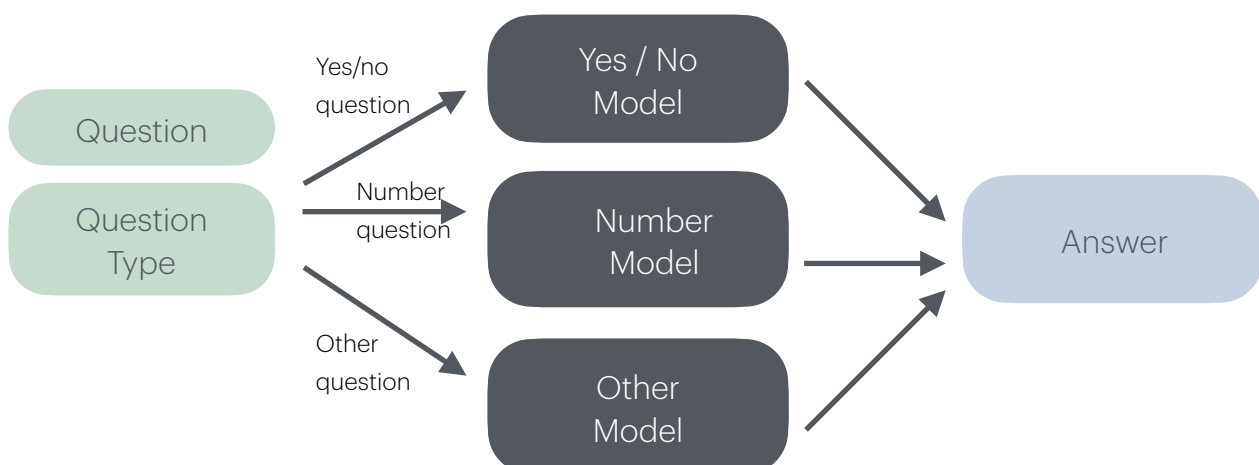
This was great news! It seemed like the only problem there was was that of an impure dataset. After cleaning, **I could see 77% validation accuracy with just 1000 samples**.

This proved that the model was working, so next, **I trained it on a 100k samples dataset**. I also used the **AdamW optimiser** instead of Adam this time, and that gave ~1% increase in accuracy:

Optimiser	Learning Rate	Epochs	Samples
AdamW	3E-05	20	100000
Time Taken (hours)	Training Accuracy	Validation Accuracy ✓	Testing Accuracy ✓
11.24	0.93	0.82	0.83

So the model was working!

This could prove to be a viable solution, **since the model was converging well for “yes/no” questions, so the model could converge for number questions as well**.



Now one problem was left, which was designing an *Other* model. As stated before, this was just a sub-problem of the original problem itself, since both in the original problem and the *Other* model problem, we don't know / have any specific format of the answer. Hence, in this case, my second approach (of clipping the output classes) could help me.

Clipped Classes

In this method, I clip the total number of classes to something much smaller than the original number of classes, which the model could then learn.

There was a huge problem of exploding classes as the number of samples increased.

Samples	Learnable Classes
100	179
10000	12632
100000	97340

Obviously, these numbers will change because we are sampling the data randomly, so the number of unique answers will also change, but they will remain in the same ballpark. In most cases, the number of answers to be learnt is more than the number of questions we have for learning. This would make it near impossible for the current model architecture to learn. So we clip the number of outputs.

For this, I did a simple analysis by maintaining a dictionary of how many times each answer occurs in the dataset. Next, I sampled various *n most common* answers from this dictionary. For $n = 2$, I was left with around 25% of the dataset, with two top most answers being "yes" or "no".

For $n = 1000$, I was left with ~64% of the dataset, i.e, 64% of all questions had common answers (only 1000 unique answers).



This was a pretty good number, since I did not lose a huge amount of data, and I also had just 1000 classes to learn (1000 classes is pretty doable since simple CNN architectures are able to perform well for imageNet classification).

So I removed any questions, answers of which did not fall in these 1000 answers set. Next, I randomly sampled questions from each of the 65 questions types to form dataset of multiple sizes such as 1k, 10k, 20k, 100k etc.

Now that the dataset was ready, it was time to try out the same architecture (Model3) again. I trained the model on a 10k dataset. Here are the results:

Optimiser	Learning Rate	Epochs	Samples
AdamW	3E-05	20	10000
Testing Time (hours)	Training Accuracy	Validation Accuracy	Testing Accuracy
1.25	0.93	0.44	0.43

This was a good sign. The validation accuracy now shot up by ~20%, and this was just with 10k samples. So, I trained the model with 100k samples, and here are the results:

Optimiser	Learning Rate	Epochs	Samples
AdamW	3E-05	20	100000
Time Taken (hours)	Training Accuracy	Validation Accuracy 	Testing Accuracy 
12.35	0.93	0.72	0.72

This was a great success! The model was performing quite well with Testing Accuracy of 72%. So in the end, I **finalised Model3** with the following parameters:

Total Trainable Parameters	Size (MB)
19,63,47,493	750

NOTE: Detailed model description is in the respective iPython notebooks.

The other performance metrics are as follow:

F1 Score	70.13
Recall	69.87
Accuracy	71.34
Precision	71.21

NOTE: Difference in accuracies in the two tables in because one was calculated using Keras Accuracy metric, while the other using sklearn API (*weighted* sum of all classes were used to calculate F1, Recall, Precision).

Now that the model was ready and was performing well, I went on for optimisation.

Optimisation

LoRa

One of the problem statements was to optimise the model as well, using LoRA fine-tuning instead of just regular fine-tuning.

So for that, the library PEFT was quite suitable as it had very intuitive APIs. But this library supported only PyTorch. Hence, I ported over Model3 to PyTorch APIs.

Next, I trained the model once using normal Fine-Tuning.

I created LoRA models of both ViT and BERT config. Here is what I got:
Here are the parameter differences:

Fine-Tuning	Trainable	Non-Trainable	Size (MB)
Default	19,63,39,778	0	750
LoRA (PEFT)	31,47,974	19,58,71,488	12

As per the PEFT output:

ViT: Trainable params: 1,339,392 || all params: 87,728,640 || trainable%: 1.5267

BERT: Trainable params: 1,339,392 || all params: 110,821,632 || trainable%: 1.2086

So going by the above statistics, we went from a reduction in size of about 750MB to about just 12MB trainable parameters. This is a drastic amount of reduction, and would be much easier to train on GPUs or CPUs.

This can lead to training the model on much larger datasets for larger batch sizes, as well as more epochs since the strain on the delegate (GPU / CPU) is just 12MB.

Due to a bit of a resource crunch, I trained both non-LoRA and LoRA models ONLY for 3 epochs and 10k sampled dataset. Since accuracy of the model was already established with proper training, I just wanted to check difference in metrics between both the models instead of the actual metrics. Here are the training statistics:

Fine-Tuning	Accuracy	Precision	Recall	F1 Score	Training Time (min)
Default	0.26	0.74	0.26	0.38	12.31 (2.5 it/s)
LoRA	0.24	0.73	0.24	0.34	12.38 (2.7 it/s)

For a little bit less accuracy, LoRA model trains (and performs) faster than default Fine-Tuned model.

Quantisation

Another popular method of optimisation is Quantisation. All of the operations in the model happen using Floating Point 32. Each weight / parameter in the model is thus a FP32 datatype which takes up 32 bits.

So in a regular 128 bit register in usual CPUs, one may be able to fit ONLY 4 numbers in parallel. So even with parallel processing on, one can process only 4 numbers per register.

Instead of training using full precision Floating Point 32, we can train using Floating Point 16. Using options in Keras API, I trained Model3 using FP16 data. Here are the results:

Optimiser	Learning Rate	Epochs	Samples
AdamW	3E-05	20	100000
Time Taken (hours)	Training Accuracy	Validation Accuracy	Testing Accuracy
8	0.91	0.70	0.69
Size (MB)			
325			

These results are pretty great, considering only a drop of 3% in accuracy, but the **size of parameters (effective) is reduced by HALF**, and the **training time reduces from ~12.35 hours to ~8 hours**, which is 0.65 times the original training time.

So in summary, we are able to process twice the amount of data as earlier, giving a reduction in training time as well as memory used.

Generative Datasets

As mentioned before, I opted to use generative datasets from the very beginning. This caused a hit to training time, but led to a HUGE improvement in memory consumption while using GPU or CPU. This is because instead of processing the dataset in one go and storing it in memory for training, I created a Generative Dataset that can process only required data (of the batch in training) and provide the processed inputs at runtime, while the rest of the data is still on disk.

This made it possible to train the GPU on a huge dataset of around 100K samples and a batch size of 16, which otherwise would not have been possible if the dataset was processed and stored in memory.

NOTE: All project resources can be found on the following link:

https://github.com/nikiiitb/Project_2/tree/vqa_custom