Final Project Report
Image Captioning Inference Service

Sili Guo (sg6163), Harsh Wani (hsw268)

1. Project Description

Image Captioning is an important topic that combines the technique of computer vision and of natural language processing. Basically, it reads the features of an image given and generate a caption that describes what picture does. This can be variously used in different applications to transfer the image into text, which can be easier understood by machine.

In our project, we build our image captioning model mainly based on two existed models on GitHub. We referenced some academic articles that describes Image Captioning model to understand the underlying principle of this technique, then we mainly follow the guideline written by Moeinh77 (GitHub username) about how to build a model for image captioning (as shown in Figure 1). Based on this model, we then tried to use the technique of hyperparameter tunning and scheduler policy changing to improve the accuracy of the model. The next step is we trained our model on native Linux, Prince, and Cloud with different types of GPUs to comparing the performance of our model under different hardware environments. The last step is to build a Flask application with our model and deploy it on Kubernetes on IBM Cloud.

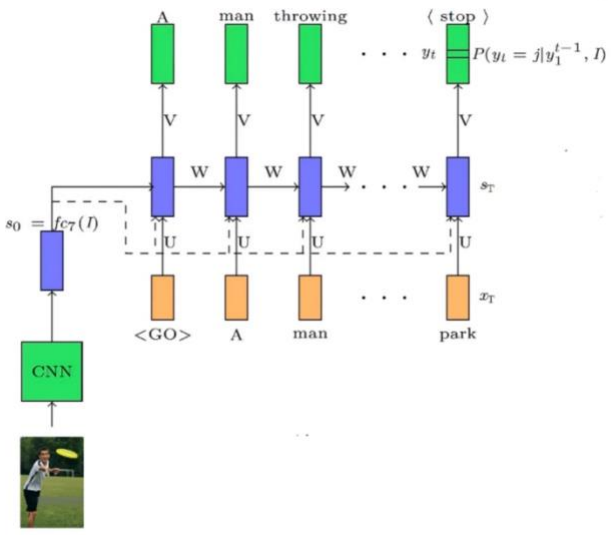

Figure 1. Image Captioning Data Generator

2. Related Works

Vinyals et al. have done a great job in building a model of image captioning and explained in the paper *"Show and Tell: A Neural Image Caption Generator".* In the paper, Vinyals et al. proposed a model to use RNN encodes a variable length input into a fixed

dimensional vector, then decode it to generate the sentence matches the input with highest probability. To build the model, they concatenated two models together to separately process the image part and text part. The first model is a CNN model that extract the feature vector of an image to create the vector. The second model is a LSTM model that takes part of sentence to predict the next word for the caption. We followed this principle and found some existing works on GitHub. We referenced two models yashk2810 and Moeinh77 (focus more on Moeinh77), to build our model.

3. Implementation

This section is separated into four stages: the final model we build, approach to improvement of accuracy, comparison on different GPUs and deployment on cloud. In this section, we will explain what we have done in each stage, and the result and evaluation of each stage will be shown in next section.

3.1 Image Captioning model

Followed by the Paper and existed works on GitHub, our model has three main parts in high level. The first part is a photo feature extractor. This is a 71-layer Xception model pre-trained on the ImageNet dataset. We have pre-processed the photos with the Xception model (without the output layer) and will use the extracted features predicted by this model as input. The second part is a Sequence Processor. This is a word embedding layer for handling the text input, followed by a Long Short-Term Memory (LSTM) recurrent neural network layer. And the last part is a Decoder. Both the feature extractor and sequence processor output a fixed-length vector. These are merged and processed by a Dense layer to make a final prediction. The progress of training is show in Figure 2.

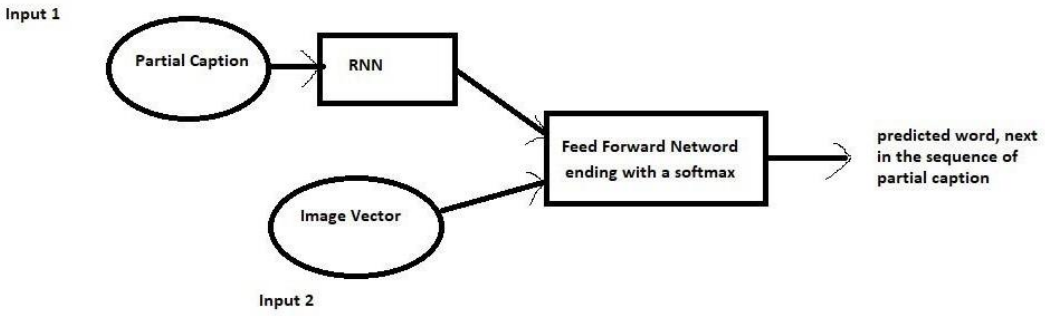

Figure 2. High-Level Architecture

3.2 Accuracy Improvement

To improve the accuracy of the model, the first approach we tried is to use hyperparameter tunning for the model. In our project, we use Grid search to find the best combination of batch size and epochs. We designed the actual experiment like this: we first make the epochs to be fixed to 25, and train with batch size equals 32, 64 and 128. We then use

the batch size with highest accuracy and increase the epochs to 50 and 75 to see which training gives us the best accuracy. We used a masked loss function to record the loss for each epoch; and to figure out when the training start to overfit, we also compare BLEU score for the caption it generated. By doing this, we found the best combination of epoch and batch size.

For the epoch and batch size that gives best accuracy, we designed two experiments that uses a learning rate scheduler with value of 0.1 and 0.01, we compare their performance with standard learning rate.

During the sequence processor part, we use different embedding layer for pre-processing text. The first one is to use the word to index method, which is a simple method that find all unique words inside the data, then sort by its similarity and then convert to a number same as its index. This technique is based on the similarity of two words. The other embedding layer we use a pre-trained dictionary from Stanford University --- GloVe. We used the one with 50 dimensions. We did the Grid search with both embedding layers to see how this make differences on the model.

3.3 Hardware Environment Comparison

We then trained our model under different hardware environments and compared the performance of training process. We first compare the performances of different GPU types on Prince with single core. The four GPUs we used are K80, P40, V100 and P100. To save the training time, we decide to train our model with batch size 32 and epochs 25 in all experiments in this section. We then run the same training process on K80 and P40 but with 4 cores and compare the result of each other and with single cores. We also compared our model by training on native Linux system and on Cloud, especially on Google Colab and a minkube instance on GCP.

3.4 Deployment on Cloud

Our last step is to build a web application with Flask and deploy on Kubernetes clusters on IBM Cloud. We used Flask package in Python to create an application with a simple HTML template for our web page. The image is transferred from web to local functions by using POST method. Then the program saves the image into a local path and read it from that path when needed. Since the image needs to be pre-processed before predict function is called, we reconstruct the model and loaded weights from training result. We then used the model loaded to predict the caption of image.

To deploy our application on Kubernetes, we first created a Docker image for the application. In the image, we put python file, Data for reconstruct the image, GloVe for pre-processing text, and the model we get from training process in it. We then push it to Docker Hub and then pulled by Kubernetes.

For the Kubernetes deployment, we create a persistent volume, a service and a deployment. The persistent volume is used for storage with capacity 1Gi. We set the access mode to be ReadWriteMany. We also create a persistent volume claim which allows a user to consume abstract storage resources. We create a node port service with TCP protocol, and set its node port to be 30,000 and target port to be 5,000. The deployment has a container named keras. It has two replica sets and will store files into path "/logs/". The actual deployments are showing in Figure 3.

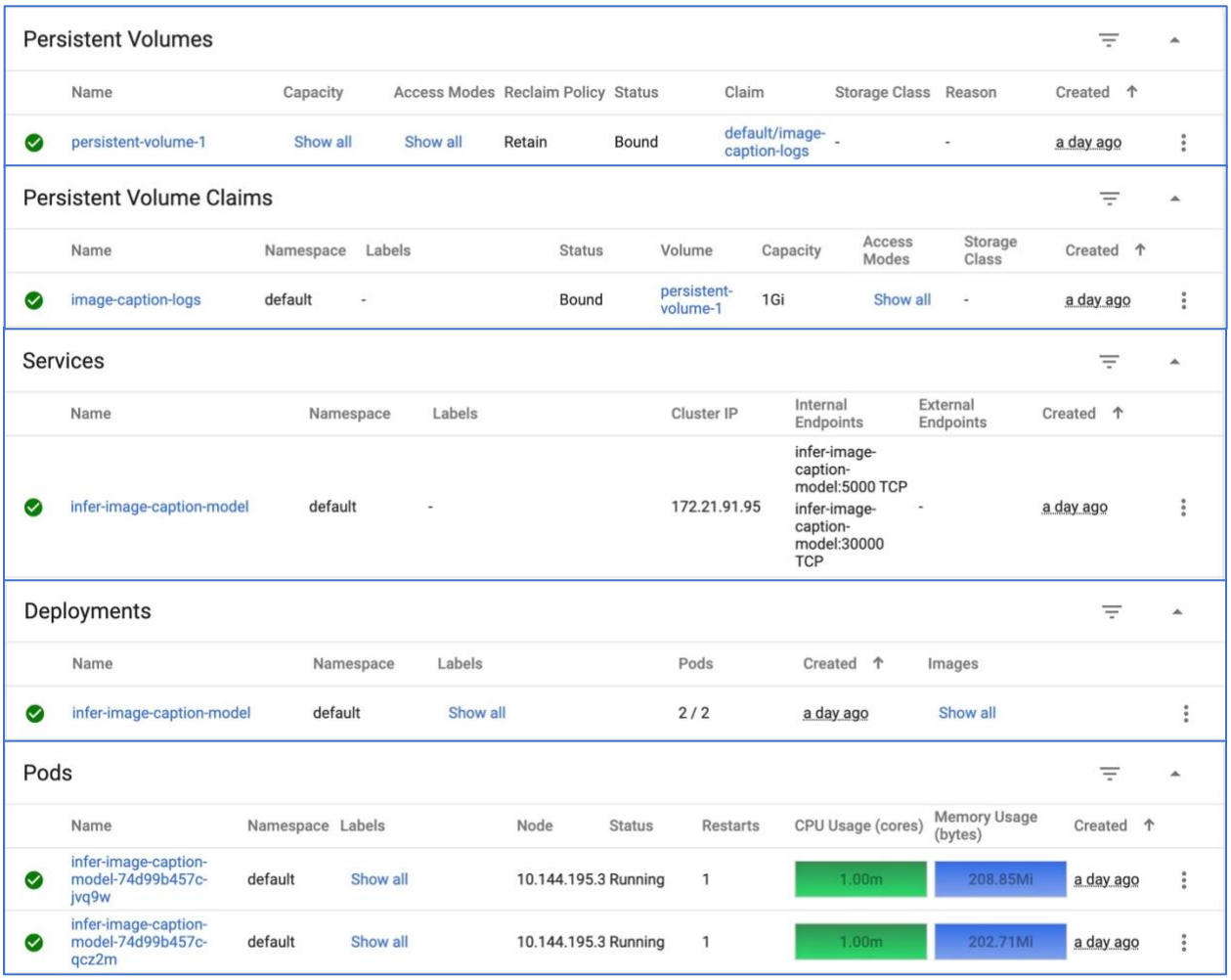

Figure 3. Deployments on Kubernetes

4. Result and Evaluation

4.1 Training Accuracy Comparison

In Table 1 and Figure 5, we are showing the result of training with GloVe using Grid Search. From the table and chart. From table, we can see the batch size 32 has the lowest loss among three experiments. For the experiments with increased epochs, the loss value is getting smaller; however, when comparing the BLEU score, the score for 75 epochs actually drops, which means overfitting happens.

Epochs	Batch-size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Loss
25	32	0.609	0.418	0.322	0.195	2.41
25	64	0.608	0.412	0.312	0.185	2.55
25	128	0.640	0.447	0.342	0.208	2.89
Epochs	Batch-Size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Loss
50	32	0.608	0.408	0.304	0.175	2.09
75	32	0.597	0.405	0.302	0.173	1.91

Table 1 Training with GloVe

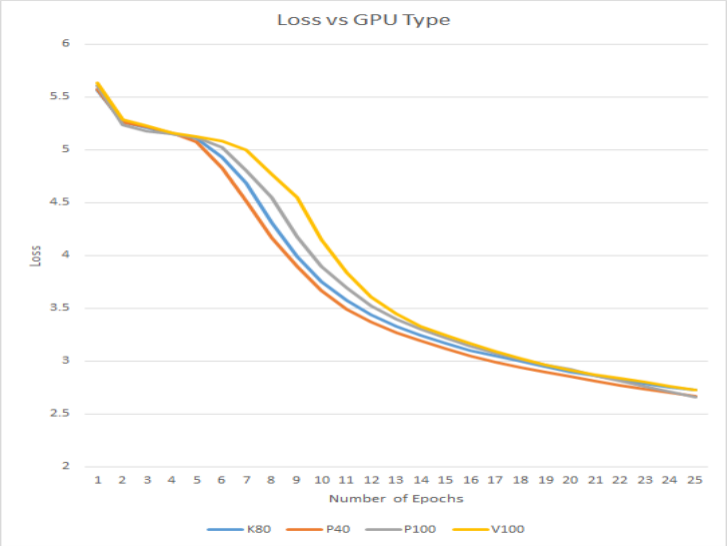

Figure 5 Training with GloVe

Table 2, Figure 6.1 and 6.2 shows the result of training with learning rate scheduler. From the result, we see it did make some improvement to the accuracy and performance when compared with standard learning rate. When we adjust the value to 0.1, it gives us a worse result due to the reason it causes the model to converge too quickly to a suboptimal solution.

Epochs	Batch-size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Loss
75 (0.01)	32	0.50	0.294	0.205	0.094	4.14
75 (0.1)	32	0.18	0.096	0.245	0.310	11.57

Table 2 – Training with Learning Rate Policy

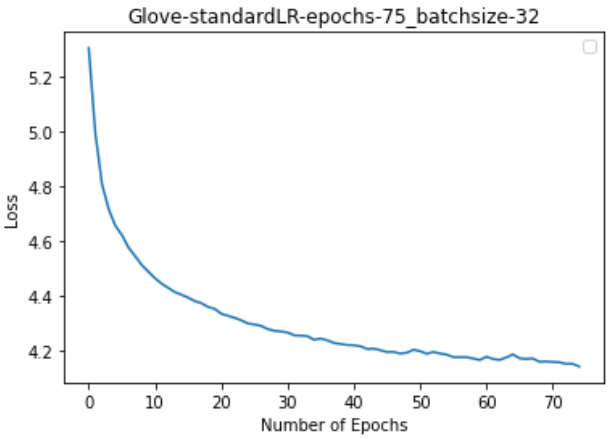

Figure 6.1 LR Policy with 0.01

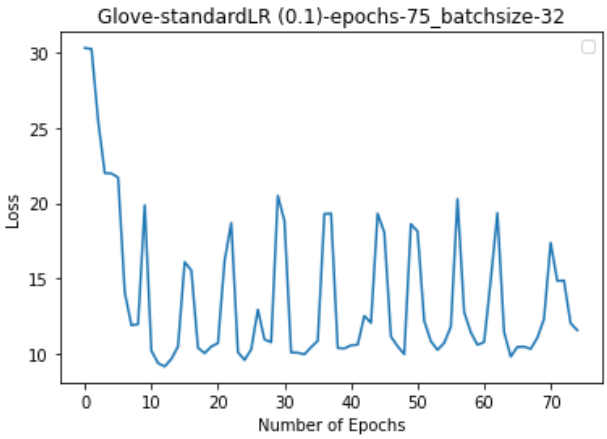

Figure 6.2 LR Policy with 0.1

Table 3 and Figure 7 shows the result of training with embedding layer using word 2 index instead of GloVe. We can see both loss and BLEU score are much lower than the embedding layer with GloVe. Similarly, for fixed 25 epochs, the batch size equals 32 gives us the best accuracy. And by increasing the epochs, we also find overfitting happens on 75 epochs.

Epochs	Batch-size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Loss
25	32	0.123	0.011	0.067	0.105	3.198
25	64	0.103	0.005	0.043	0.073	3.410
25	128	0.101	0.012	0.003	0.009	3.564
Epochs	Batch-Size	BLEU-1	BLEU-2	BLEU-3	BLEU-4	Loss
50	32	0.107	0.014	0.004	0.0104	2.82
75	32	0.101	0.013	0.003	0.0101	2.69

Table 3 Training without GloVe

Further, we tried to mitigate the issue of overfitting with increasing the number of epochs and designing the model with batchnorm layers and dropout techniques. But this did not give us satisfactory results and hence, we did not explore this further.

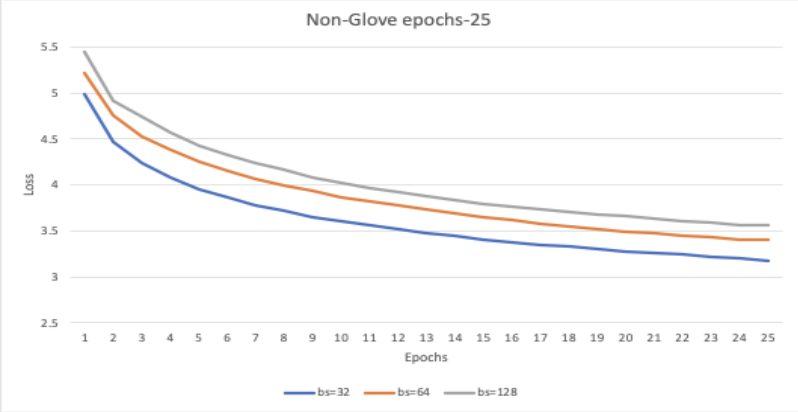

Figure 7 Training without GloVe

4.2 Hardware Environments Comparison

Figure 8.1 and 8.2 shows the comparison result on different GPUs (single core) on Prince. We trained with batch sizes equals to 32, 64 and 128, but only 25 epochs as it was computationally relatively less expensive. Figure 8.1 focus on the performance on different batch sizes. From chart we can see V100 did much better than other three GPUs under all conditions; while P100 did worst under all conditions. There is no surprise that the GPU has better performance on batch size 32 will still have better performance with different batch sizes. In Figure 8.2 we group the result based on different GPU types, we can see when increasing batch size to 64, all of them have better performance; however, when further increasing batch size to 128, except for V100, all other GPU actual takes longer for the training process. We guess this may happened because with a higher batch size, the model become hard to reach the minimum point.

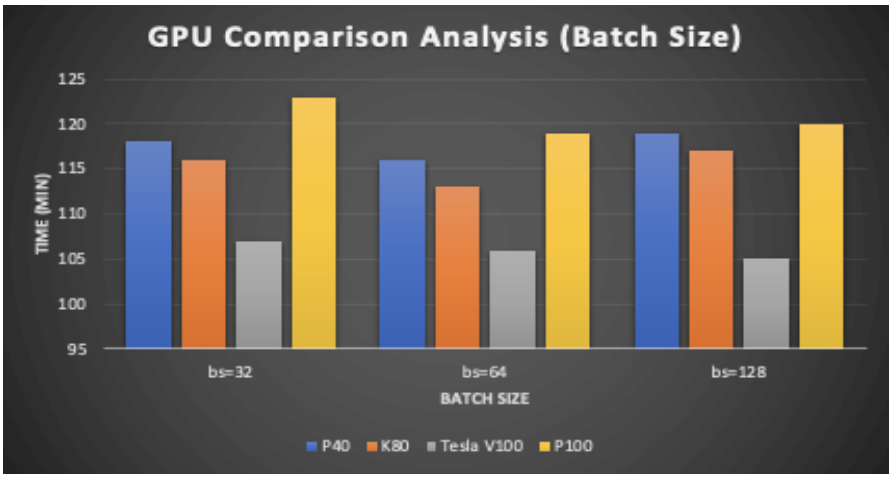

Figure 8.1 Single-core GPU Comparison (Group with batch size)

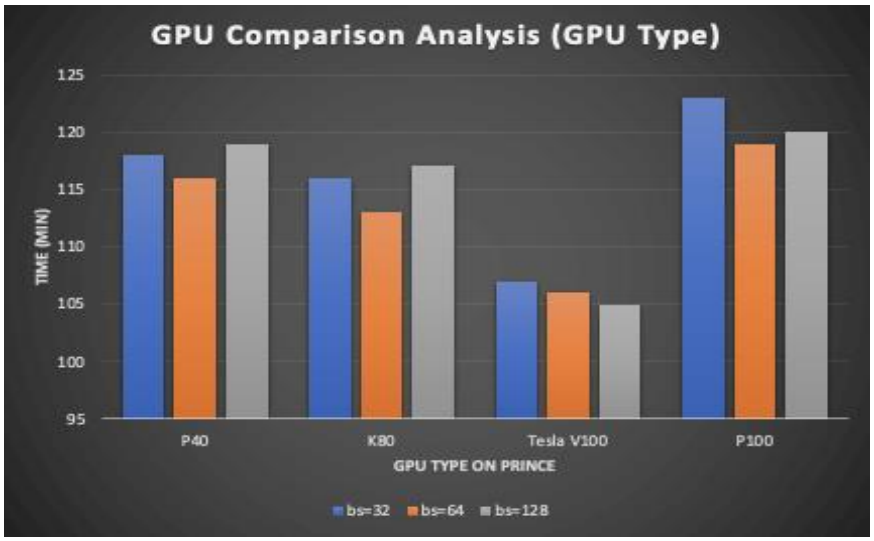

Figure 8.2 Single-core GPU Comparison (Group with GPU type)

Figure 8.3 shows the performance that training with multi-cores. We used K80 and P40 on Prince each with 4 cores. Compare with performance of single core GPUs, which is always higher than 110 minutes, the performance of multi-core GPUs is largely improved, which is always between 40 to 50 minutes. However, it did not decrease its time to one forth as we expected. We think this may be caused by delay in network. With batch size increasing, we realize the training process takes longer, and this may also because of the reason of delay. The performance of different GPU shows similar result as single core result.

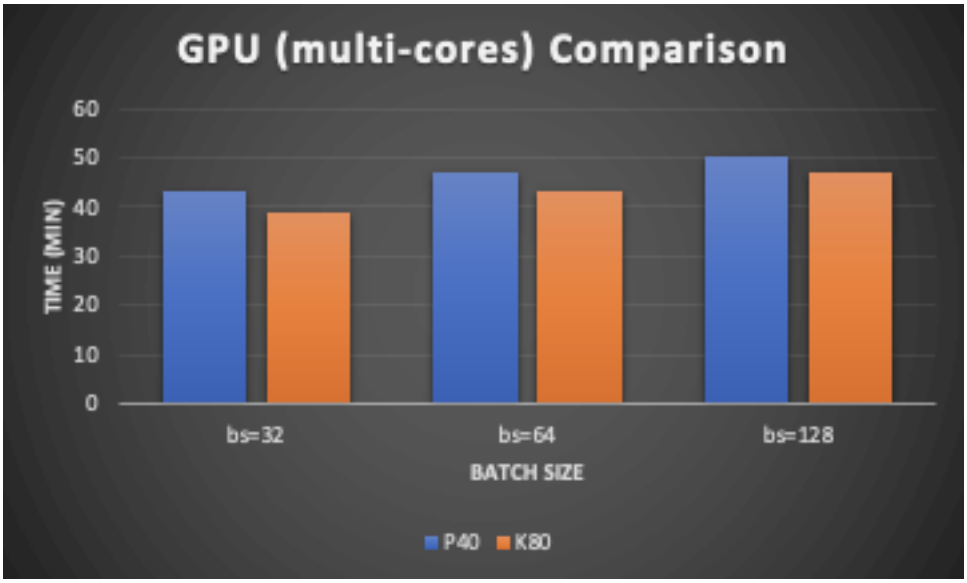

Figure 8.3 Multi-core GPU Comparison

4.3 Bare Metal and GCP Comparison

We didn't get great result for this set of comparison due to the problem of we cannot set same hardware environment on both. So, in Table 4, we record the experiment result we have and hopefully in future days if we have chance to setup this experiment with same GPUs, we can complete this. The result shows the training performance on local Linux with 2080Ti, and on GCP with V8 CPUs.

	Loss	Time (min)
GCP	3.24	108
Bare Metal	3.18	56

Table 4. Bare Metal and GCP Comparison

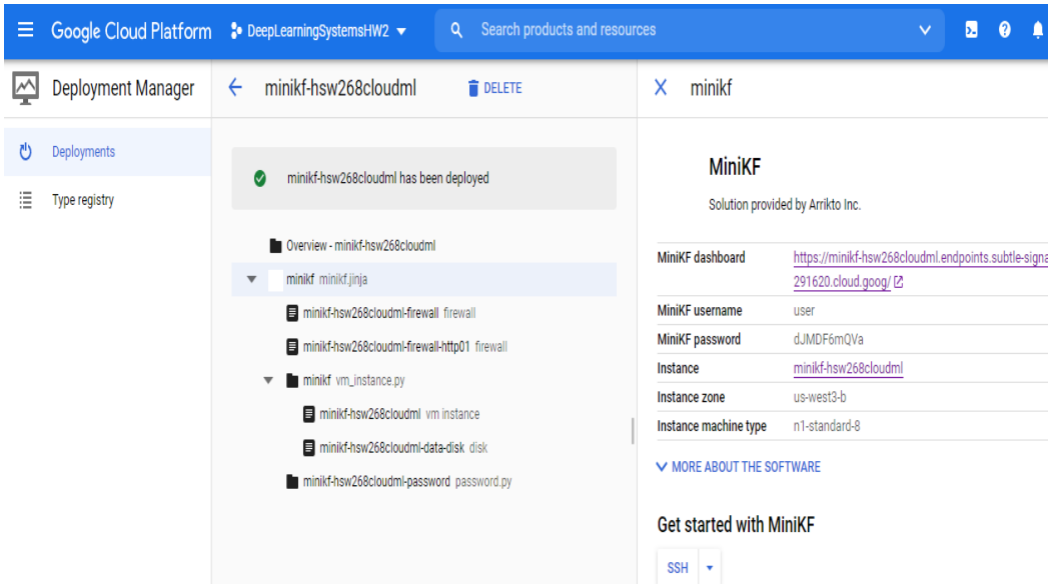

Figure 9. Deployment of MiniKube Instance on GCP

5. Challenges

In our project, we faced a lot of challenges during the process in each stage. The first challenge we had is to make the model work. The code on GitHub we originally based on did not work because the Keras and TensorFlow version the code used are outdated and some of

functions no longer exists. To modify the model, we read the paper and referenced another model that finally make the training process work.

Another challenge we faced during the comparison stage is it is hard to create an exact same hardware environment for all of our experiments. So, to solve this problem, we tried to minimize the variation between our experiments. For example, we used same GPUs on Prince to test multi-core effects. But we didn't make all of our experiments in same environment, especially the experiment for GCP and Bare Metal, and we hope we can find a solution to complete this comparison in the future.

Also, building an application on Flask is also a challenge for us. Since we are both programmers focus on backend side, we don't have many experiences working on web applications. To make the result show on browser, we spent some time learning how to use POST and GET and to transfer images between web server and local functions. We originally plan to also show images for testing on the browser but due to the reason of lack of knowledge in this field, we didn't make that function works. If there's more time, we hope we can also build a more user-friendly UI for our project.

6. Conclusion and Future Work

In our project, we have achieved our original goal of building a image captioning application. We build our model successfully based on some existing works, and we have tried to improve the accuracy in three aspects: tuning the hyperparameters to find out the best accuracy of epochs and batch sizes, adding learning rate schedule and using different embedding layer with and without GloVe. We also successfully deployed our application on Kubernetes on IBM Cloud using Docker and Flask.

From the comparison result we have shown in section 4, we are able to find out the best combination of epochs and batch sizes using Grid search. It shows us that increasing the batch size and learning rate will decrease the loss but should be careful about the point it starts to overfit. Also, we have realized that either increasing the batch size or decaying the learning rate may reduce the loss, but if we increase the batch size and decay the learning rate both at same time, the result did not improve much. Hence, decaying the learning rate during the training while keeping the batch-size constant serves best generally. We are also able to see the performance of different GPU types and the effects caused by multi-cores. With multi-cores the training time did not reduce as much as we thought due to the network delay. With multi-core systems, the system achieves parallelism but must pay for communication costs. There is sharing of knowledge between multiple GPUs and between various nodes (cores)

In the future, we plan to further increase the accuracy of our model by add an attention layer into the model. By using attention, the model can focus more on the important part of the caption text in datasets, which makes the prediction more reasonable.

7. Demo Examples

7.1 Demo 1

Welcome to our Image Captioning Application

Select a file to upload

Choose File | No file chosen

Submit

Result: a person kayaks in rough water

(Pictures will not show in browser, show here to compare with caption generated)

7.2 Demo 2

Welcome to our Image Captioning Application

Select a file to upload

Choose File No file chosen

Submit

Result: a young boy runs through a field

8. Reference

[1] Vinyals, Oriol, et al. "Show and tell: A neural image caption generator." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015.
[2] yashk2810, https://github.com/yashk2810/Image-Captioning.git
[3] Moeinh77, https://github.com/Moeinh77/Image-Captioning-with-Beam-Search.git