Low Resolution to High Resolution Image Enhancement

CMPE- 297 SEC 49- Special Topics

Project Report

By
Srilalitha Veerubhotla
Shailesha Prasad Maganahalli
Atul Shah
Shreyus Puthiyapurail

Table of Contents

Significance of the project

Image super resolution can be defined as increasing the size of small images while keeping the drop-in quality to minimum or restoring high resolution images from rich details obtained from low resolution images. This problem is quite complex since there exist multiple solutions for a given low resolution image. This has numerous applications like satellite and aerial image analysis, medical image processing, compressed image/video enhancement etc.

In our project, we enhance low resolution images by applying deep network with adversarial network (Generative Adversarial Networks) to produce high resolutions images.

Goal of the project is to reconstruct super resolution image or high-resolution image by up-scaling low-resolution image such that texture detail in the reconstructed SR images is not lost.

Dataset

We used CIFAR-10 dataset

The CIFAR-10 dataset consists of 60000 32x32 color images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class. The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class.

Here are the classes in the dataset, as well as 10 random images from each:

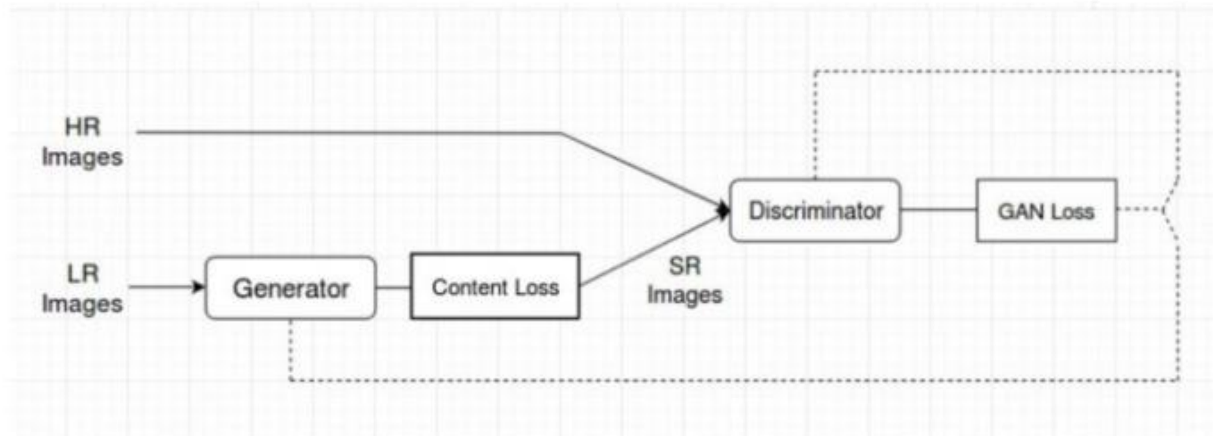
Models

We performed two different models to choose the best one for our application

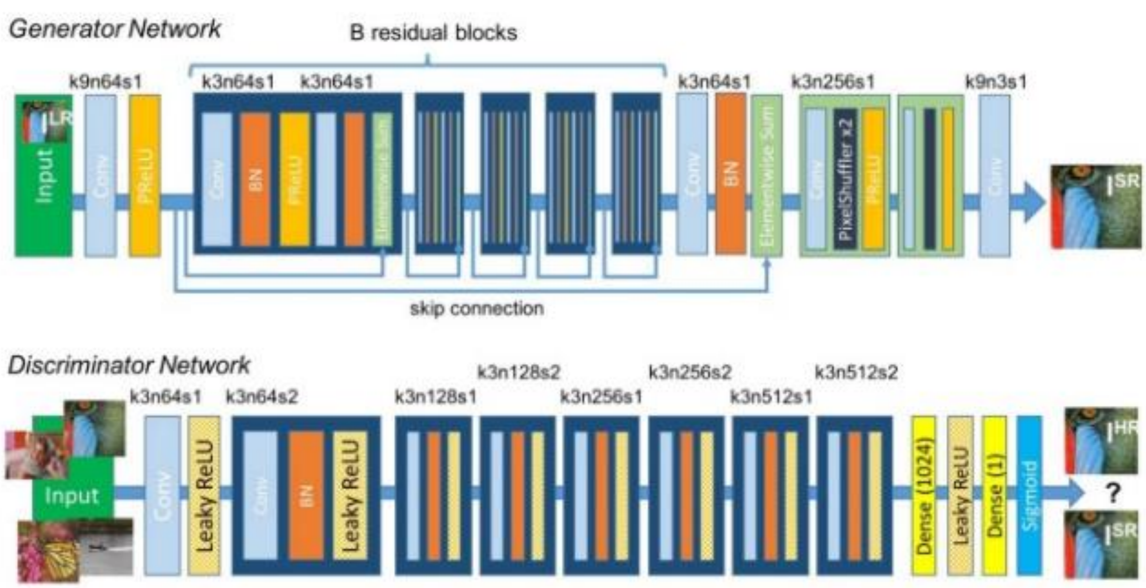
1. SRGAN
2. SRCNN

After a thorough research and implementation, we started working on SRGAN which is performing better than SRCNN.

SRGAN Architecture



Generator and Discriminator Architecture



Model Training

Training procedure is shown in following steps:

- We process the HR(High Resolution) images to get down-sampled LR(Low Resolution) images. Now we have both HR and LR images for training data set.
- We pass LR images through Generator which up-samples and gives SR(Super Resolution) images.
- We use a discriminator to distinguish the HR images and back-propagate the GAN loss to train the discriminator and the generator.

Model Parameters

1. DATASET USED : CIFAR 10
2. TRAINING IMAGES : 5000
3. TEST IMAGES : 800
4. LOSS FUNCTIONS : ADVERSARIAL LOSS(BINARY CROSS ENTROPHY), CONTENT LOSS (VGG LOSS)
5. OPTIMIZER : ADAM with
    1. Learning Rate = 1E-4
    2. beta_1=0.9
    3. beta_2=0.999
    4.epsilon=1e 08
6. EPOCHS : 2
7. HARDWARE USED : COLAB GPU
8. Downscale Factor = 4
9. BATCH SIZE : 78

```python
class Generator(object):

    def __init__(self, noise_shape):

        self.noise_shape = noise_shape

    def generator(self):

            gen_input = Input(shape = self.noise_shape)

            model = Conv2D(filters = 64, kernel_size = 9, strides = 1, padding = "same")(gen_input)
            model = PReLU(alpha_initializer='zeros', alpha_regularizer=None, alpha_constraint=None, shared_axes=[1,2])(model)

            gen_model = model

        # Using 16 Residual Blocks
            for index in range(16):
                model = res_block_gen(model, 3, 64, 1)

            model = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = "same")(model)
            model = BatchNormalization(momentum = 0.5)(model)
            model = add([gen_model, model])

            # Using 2 UpSampling Blocks
            for index in range(2):
                model = up_sampling_block(model, 3, 256, 1)

            model = Conv2D(filters = 3, kernel_size = 9, strides = 1, padding = "same")(model)
            model = Activation('tanh')(model)

            generator_model = Model(inputs = gen_input, outputs = model)

            return generator_model
```

Code Snippet for Generator

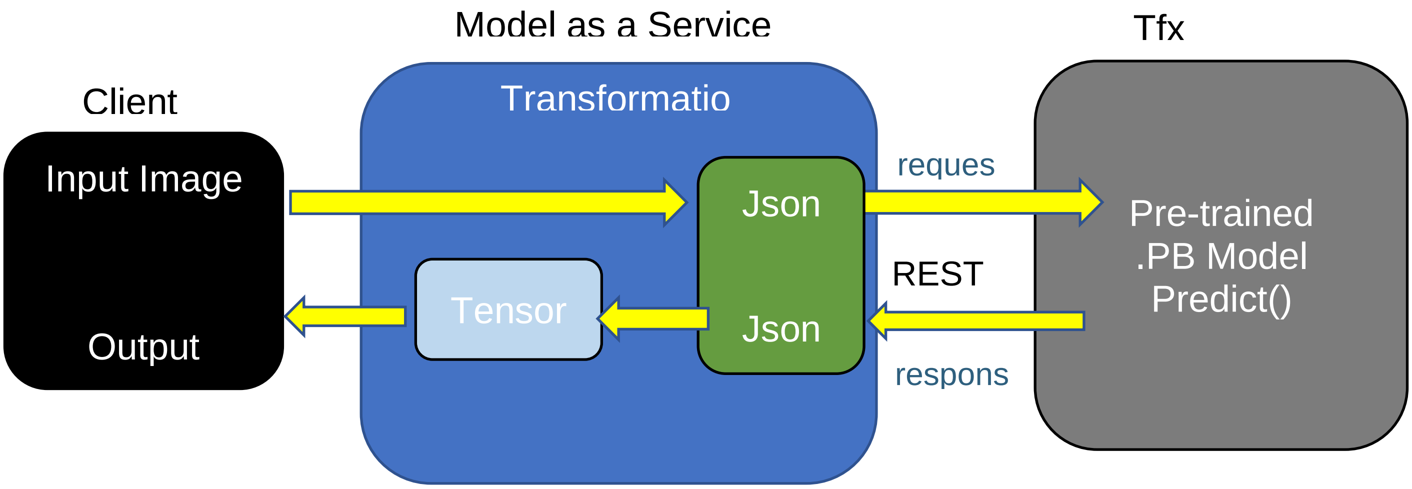
```python
class Discriminator(object):

    def __init__(self, image_shape):

        self.image_shape = image_shape

    def discriminator(self):

        dis_input = Input(shape = self.image_shape)

        model = Conv2D(filters = 64, kernel_size = 3, strides = 1, padding = "same")(dis_input)
        model = LeakyReLU(alpha = 0.2)(model)

        model = discriminator_block(model, 64, 3, 2)
        model = discriminator_block(model, 128, 3, 1)
        model = discriminator_block(model, 128, 3, 2)
        model = discriminator_block(model, 256, 3, 1)
        model = discriminator_block(model, 256, 3, 2)
        model = discriminator_block(model, 512, 3, 1)
        model = discriminator_block(model, 512, 3, 2)

        model = Flatten()(model)
        model = Dense(1024)(model)
        model = LeakyReLU(alpha = 0.2)(model)

        model = Dense(1)(model)
        model = Activation('sigmoid')(model)

        discriminator_model = Model(inputs = dis_input, outputs = model)

        return discriminator_model
```

Code Snippet for Discriminator

```
discriminator_loss : 0.346830
gan_loss : [0.07546138018369675, 0.07462482154369354, 0.8365601301193237]
In plot test generated images
800
Model saved
```

Code Snippet for Loss

Model as a Service



Model is hosted as a service on Tfx Serving. The web application client passes the image as a collection of pixels in the form of json object to the model. The model accepts the json object and processes the image using SRGAN algorithm. The model returns the super resolution image in the form of json object. The json object is then converted to NumPy array which is further processed into TensorFlow object and later saved as JPEG super resolution image.

The following code snippets is walkthrough for image processing for model as a service.

Model Input Processing

model.get_prediction(image_path) **# Step 1. Call the model from web client**

image = Image.open("Original Image.jpg") **# Step 2. Access the image and store in PIL format**

im = np.asarray(image) **# Step 3. Convert image above as a numpy array**

data = json.dumps({"instances": im.tolist()}) **# Step 4. Create a json object from numpy array as key value pair to be passed to the model**
rv = requests.post(SAVED_MODEL_PATH, data=data) **#Step 5. REST API call to the model with json object**

Model Output Processing

response = json.loads(rv.text) **# Step 1. Extract the json from response variable** response_string = response['predictions'][0] **# Step 2. Extract values corresponding to “Predictions” key**

sr_image = tf.image.convert_image_dtype((np.asarray(response_string)),

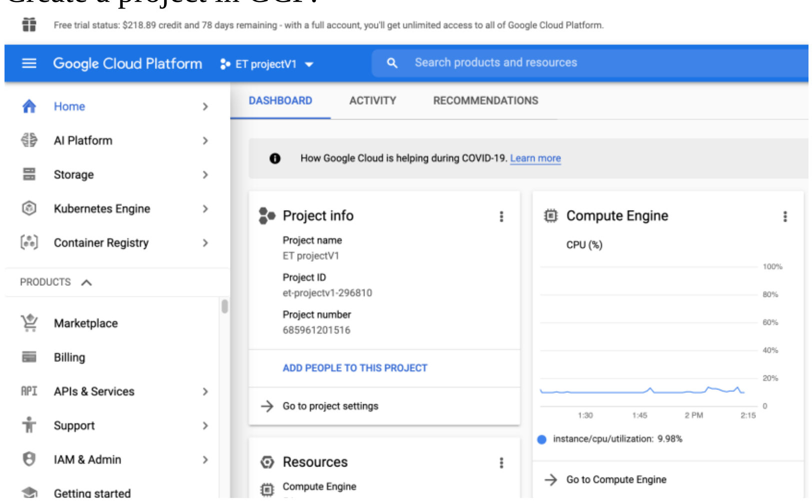
dtype=tf.float32, saturate=True) **# Step 3. Convert numpy array to a tensor**

save_image(sr_image, filename=filename) **# Step 4. Convert tensor to an Image and return to the client**

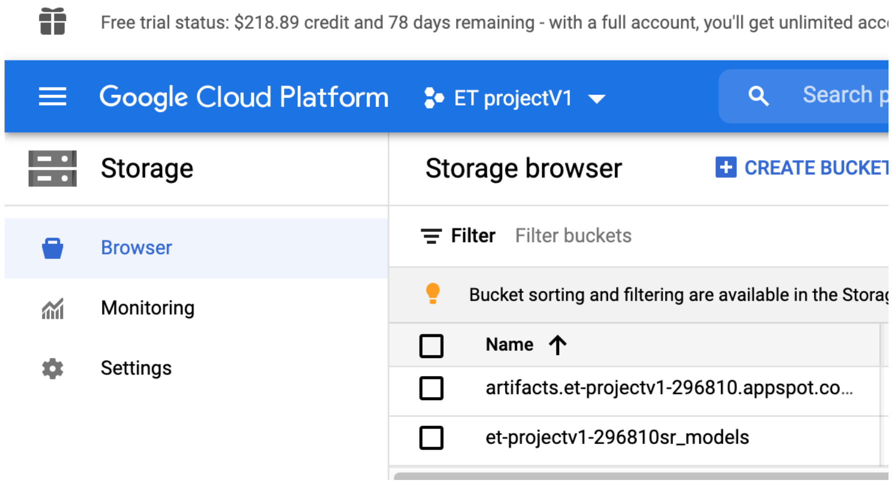
Model Deployment

Once the SRGAN model was trained and tested, the deployment ready model was saved in the form of .pb file. We decided to make the model servable by creating it as service using Kubernetes on google cloud platform. Below are the detailed steps involved in deployment of the saved model.

1. Set up the GCP cloud environment:
    - Created a google cloud environment and acquired a credit of $300 good enough to use the GCP for our requirement of model deployment.
    - Create a project in GCP.



    - Setup the storage bucket where our model was saved.



2. Create docker image for model serving:
    - In-order to make the model servable, pull the tensorflow serving using the below command.
      # docker pull tensorflow/serving
    - Run the tensorflow serving to create a serving base for our model.
      # docker run -d --name serving_base tensorflow/serving

    - Copy our saved model to the serving base in "/models/model"
      # docker cp /home/shreyus_puthi/srgan serving_base:/models/model

    - Commit the lates model to the serving base
      # docker commit serving_base model:latest

    - Run the latest model to deploy it as a docker image on port 8501
      # docker run -d --name=tfgan -p 8500:8500 -p 8501:8501 model:latest

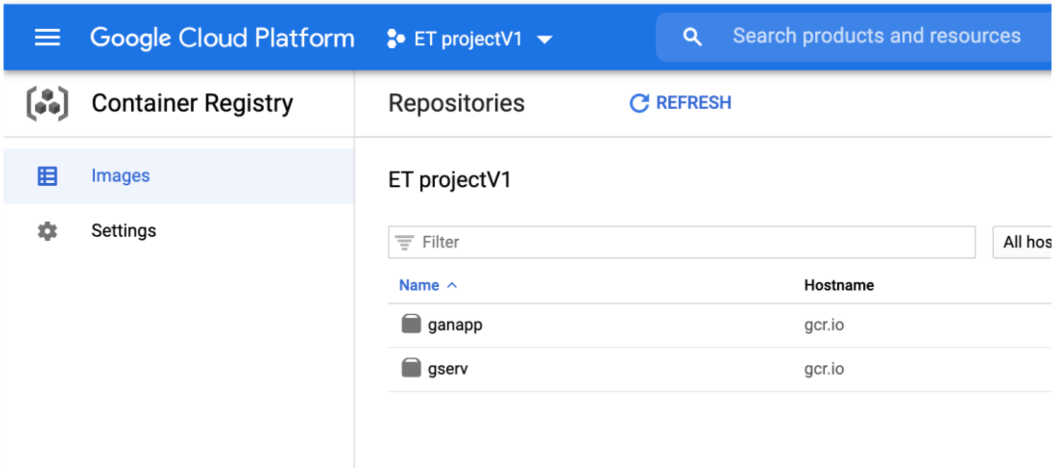
    - Test the model using curl command
      # curl http://localhost:8501

   Once the model is up and running deploy it on Kubernetes.

3. Deploy the model on Kubernetes as a service.

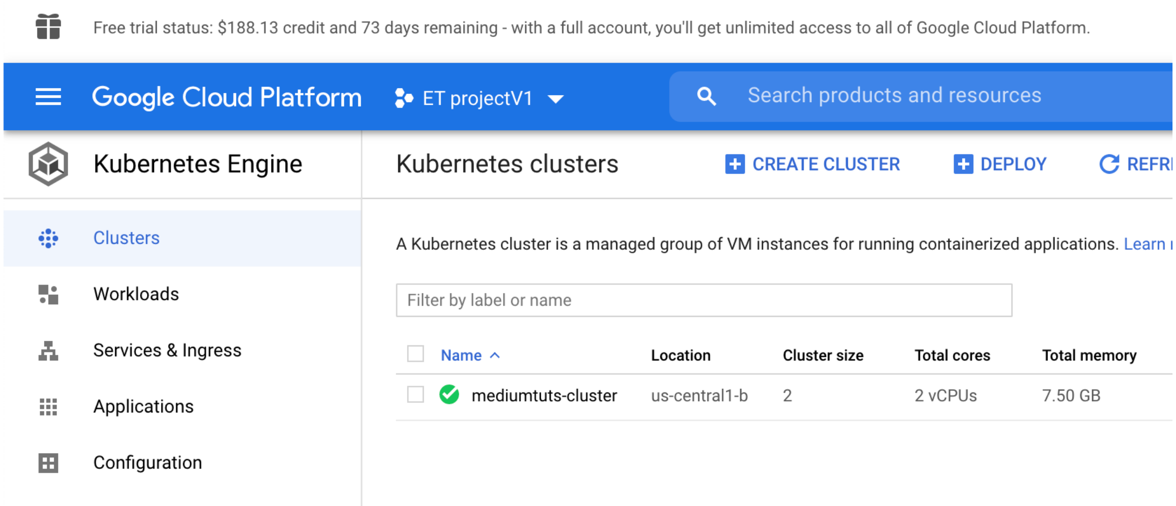
    - Tag the model to the gcp project created above.
      # docker tag model gcr.io/et-projectv1-296810/gserv:v1

    - Push the model to the image container of gcp.
      # docker push gcr.io/et-projectv1-296810/gserv



- Login to the project
  # gcloud auth login --project et-projectv1-296810

- Create Kubernetes cluster in GCP with required number of nodes.
  # gcloud container clusters create mediumtuts-cluster --zone us-central1-c --num-nodes 2



- Set the to the cluster created.
  # gcloud config set container/cluster mediumtuts-cluster

- Create the yaml file for the model to be served on 8501 port as below, specifying image path in the container registry.
  # cat tfgan_model.yaml

```
shreyus_puthi@cloudshell:~ (et-projectv1-296810)$ cat tfgan_model.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: tfmodel
spec:
  replicas: 3
  selector:
    matchLabels:
      app: tfmodel
  template:
    metadata:
      labels:
        app: tfmodel
    spec:
      containers:
      - name: tfmodel
        image: gcr.io/et-projectv1-296810/gserv@sha256:b5c84539ba1ae2a
        ports:
        - containerPort: 8501
---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: model-service
  name: model-service
spec:
  ports:
  - port: 8501
    targetPort: 8501
  selector:
    app: tfmodel
  type: LoadBalancer
```

- Deploy the model using the below command.
  # kubectl create -f tfgan_model.yaml.

  This creates the pods, deployment and services.
  Service details of model-service:

```
shreyus_puthi@cloudshell:~/.kube/cache/http (et-projectv1-296810)$ kubectl get svc
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
gan-service     LoadBalancer   10.99.254.255   35.223.8.217     5000:30788/TCP   98m
kubernetes      ClusterIP      10.99.240.1     <none>           443/TCP          8d
model-service   LoadBalancer   10.99.255.110   35.224.126.101   8501:32568/TCP   112m
shreyus_puthi@cloudshell:~/.kube/cache/http (et-projectv1-296810)$
```

  Pod details:

```
shreyus_puthi@cloudshell:~ (et-projectv1-296810)$ kubectl get pods
NAME                          READY   STATUS    RESTARTS   AGE
gan-server-64fd9749c-4h6tn    1/1     Running   0          4d23h
gan-server-64fd9749c-8hhfm    0/1     Evicted   0          4d23h
gan-server-64fd9749c-b22fk    1/1     Running   0          4d23h
gan-server-64fd9749c-mh2gm    1/1     Running   0          4d22h
tfmodel-79464f4446-c6hdp      1/1     Running   0          5d
tfmodel-79464f4446-w77g5      1/1     Running   0          5d
tfmodel-79464f4446-xzl9h      1/1     Running   0          5d
```

  Deployment details

```
shreyus_puthi@cloudshell:~ (et-projectv1-296810)$ kubectl get deployments
NAME          READY   UP-TO-DATE   AVAILABLE   AGE
gan-server    3/3     3            3           4d23h
tfmodel       3/3     3            3           5d
```

- Test the deployed model (basic test) by running the curl command or on web url

  # curl http://35.224.126.101:8501/v1/models/model


```
{
 "model_version_status": [
  {
   "version": "1",
   "state": "AVAILABLE",
   "status": {
    "error_code": "OK",
    "error_message": ""
   }
  }
 ]
}
```


Web Application Deployment

The web application is created for model inference. The application is deployed on Kubernetes as a service.

1. Creating docker image of web app.

- Create a folder App with all the python modules required for web app.

```
shreyus_puthi@cloudshell:~/web/App (et-projectv1-296810)$ ls
 app.py     model_b64.py    model_old.py      model.py                 my-deployment.yaml       __pycache__   static
 hello.py   model_json.py   model_opencv.py   my-cip-service.yaml     'Original Image.jpg'      rand.png      Template
```

- Create the docker file and requirement file outside the App folder.

```
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$ ls
2.0  App  Dockerfile  ganapp.yaml  gan.yaml  out  requirements.txt
```

- Create requirements file. The requirement file consists all the required installations required for the web application.

```
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$ cat requirements.txt
tensorflow==2.3.1
tensorflow-hub
flask
flask-bootstrap
requests
pillow
grpcio
grpcio-tools
matplotlib
```

- Create docker file to build a docker image.

```
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$ cat Dockerfile
FROM python:3.7

WORKDIR /home/shreyus_puthi/web

COPY requirements.txt /tmp/

RUN ls /tmp

# upgrade pip and install required python packages
RUN pip3 install -r /tmp/requirements.txt

RUN pip3 install --upgrade tensorflow-hub

# copy over our app code
COPY App/ .

CMD [ "python3", "./app.py" ]
```

- Create a docker image for the application.
  # docker build -t App .

2. Once you have the docker image follow the step 3 from the above “Deploy the model on Kubernetes as a service” to deploy the app as a service on Kubernetes.

   - Create the yaml file for the app with port 5000.

```
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$ cat ganapp.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: gan-server
spec:
  replicas: 3
  selector:
    matchLabels:
      app: gan-server
  template:
    metadata:
      labels:
        app: gan-server
    spec:
      containers:
      - name: gan-container
        image: gcr.io/et-projectv1-296810/ganapp@sha256:f6aa9f963e0f6ab
        ports:
        - containerPort: 5000
---
apiVersion: v1
kind: Service
metadata:
  labels:
    run: gan-service
  name: gan-service
spec:
  ports:
  - port: 5000
    targetPort: 5000
  selector:
    app: gan-server
  type: LoadBalancer
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$
```

- Once the app is deployed, it will be available as a service .

```
shreyus_puthi@cloudshell:~/web (et-projectv1-296810)$ kubectl get svc
NAME            TYPE           CLUSTER-IP      EXTERNAL-IP      PORT(S)          AGE
gan-service     LoadBalancer   10.99.254.255   35.223.8.217     5000:30788/TCP   5d
kubernetes      ClusterIP      10.99.240.1     <none>           443/TCP          13d
model-service   LoadBalancer   10.99.255.110   35.224.126.101   8501:32568/TCP   5d
```

Web Application UI

Code Repository

Link to the code:
https://github.com/Image-Enhancement-Team-Invincibles/Advanced_Deep_Learning

Application URL: http://35.223.8.217:5000

Model URL: http://35.224.126.101:8501/v1/models/model

References

[1] Arslan, M. (2020, April 30). Deploying Deep Learning Models using TensorFlow Serving with Docker and Flask. Retrieved December 13, 2020, from https://towardsdatascience.com/deploying-deep-learning-models-using-tensorflow-serving-with-docker-and-flask-3b9a76ffbbda

[2] G. (n.d.). Use TensorFlow Serving with Kubernetes : TFX. Retrieved December 13, 2020, from https://www.tensorflow.org/tfx/serving/serving_kubernetes

[3] Deepak112. (n.d.). Deepak112/Keras-SRGAN. Retrieved December 13, 2020, from https://github.com/deepak112/Keras-SRGAN