Operationalization

Azure Machine Learning Workbench has an integrated CLI which allows to deploy a model both locally and in a Kubernetes cluster, however for this project we used it just for the local deployment. In the score.py file we used a local image, the goal was to have a simple model deployed in a web service. Here's how score.py looks like:

```
def init():
    global model
    model = load_model("proc/pst/cntk.model")
    node  = model.find_by_name("poolingLayer")
    model = combine([node.owner])

    global svmLearner, svmBias, svmWeights
    svmLearner    = loadFromPickle("proc/pst/svm.np")
    svmBias    = svmLearner.base_estimator.intercept_
    svmWeights = np.array(svmLearner.base_estimator.coef_[0])

def get_image_from_blob(url):
    return "pass"

def run(input_df):
    evaluation_image = "D:/Datasets/foreground_segmented/12083123sj.quick.jpg" #get_image_from_blob(input_df["url"])
    queryImg = imread(evaluation_image)
    refImgInfos = loadFromPickle("proc/pst/imgInfosTest.pickle")
    ImageInfo.allFeatures = loadFromPickle("proc/pst/features.pickle")

    imgPadded = imresizeAndPad(queryImg, 224, 224, padColor = [114,0,0])
    arguments = {
        model.arguments[0]: [np.ascontiguousarray(np.array(imgPadded, dtype=np.float32).transpose(2, 0, 1))], # convert to CNTK's HWC format
    }

    dnnOut = model.eval(arguments)
    queryFeat = np.concatenate(dnnOut, axis=0).squeeze()
    queryFeat = np.array(queryFeat, np.float32)

    dists = []
    for refImgInfo in refImgInfos:
        refFeat = refImgInfo.getFeat()
        dist = computeVectorDistance(queryFeat, refFeat, distMethod,
        True, svmWeights, svmBias, svmLearner)
        dists.append(dist)

    sortOrder = np.argsort(dists)
    top10 = sortOrder[:10]
    top10_images = [refImgInfos[index].getImgPath(imgDir) for index in top10]
    identifiers = [os.path.split(image_path)[1].split("_")[0] for image_path in top10_images]
    return json.dumps(identifiers)


def main():
    init()
    identifiers_json = run("")
    return identifiers_json

if __name__ == "__main__":
    json_identifiers = main()
    sys.exit(json_identifiers)
```

The second step was to deploy the model on a Linux DSVM, but in that case we couldn't use Workbench because at the time of this writing it's not available for Linux machines. Thus, we wrapped this up using Flask and ran the app on the DSVM.

The final step was the cluster deployment: our goal was to use the GPUs to train the model on Azure and, even though with Workbench we can actually create a Kubernetes cluster with any kind of VM, at the moment of this writing NVIDIA drivers are not automatically installed. That's the reason why we decided to use Azure Container Service V2 and we created a cluster with GPUs enabled using few commands in the Azure CLI:

az acs create --orchestrator-type kubernetes --name <your-acs-name> --resource-group <your-resource-group> --generate-ssh-keys --location <your-location> --agent-vm-size Standard_NC6

It's important to choose a region where N-Series VM are available and also ACS V2 is available. With the following commands we can verify that GPUs are actually enabled in our VMs.

az acs kubernetes get-credentials --name <your-acs-name> --resource-group <your-resource-group>

kubectl get nodes

kubectl describe node <node-name>

Finally, in order to use the GPUs to train our model, we had to expose the NVIDIA drivers from the host to the container and to specify in the job template how many of them we actually need. Here's how the job template should look like:

```yaml
apiVersion: batch/v1
kind: Job # We want a Job
metadata:
  name: nvidia-smi
spec:
  template:
    metadata:
      name: nvidia-smi
    spec:
      restartPolicy: Never
      volumes: # Where the NVIDIA driver libraries and binaries are located on the host
      - name: bin
        hostPath:
          path: /usr/lib/nvidia-384/bin
      - name: lib
        hostPath:
          path: /usr/lib/nvidia-384
      containers:
      - name: nvidia-smi
        image: nvidia/cuda # Which image to run
        command:
          - nvidia-smi
        resources:
          limits:
            alpha.kubernetes.io/nvidia-gpu: 1 # Requesting 1 GPU
        volumeMounts: # Where the NVIDIA driver libraries and binaries should be mounted inside our container
        - name: bin
          mountPath: /usr/local/nvidia/bin
        - name: lib
          mountPath: /usr/lib/nvidia
```

Additional information about how to use the GPUs with Kubernetes can be found here.

Image Similarity Web Application

The brains of image similarity are the Machine Learning algorithms that compare the existing catalogue of images with uploaded images, and the Web App acts as the front end to accept and show the images.

The diagram below shows the various components of the data pipeline. While in certain cases, it may be possible to directly expose the ML Query component to the client and raw uploaded image, but for more accurate results, we may need to pre-process the image or enrich the outgoing message. It may also be required to limit the size of the image. This pipeline is built to be flexible enough to support such scenarios.

The exchange of messages is primarily through the use of queues, where notifications are sent to the various queues and the result is stored in Azure Cache. There is shared information about the queues, cache and the storage account between the components.

A distributed cache was chosen versus a more durable store like a database (SQL Server, Cosmos DB) considering the longevity of query would be 30 seconds or lower.

Components

The components include:

Azure Web App (ASP.NET Core 2.0 /MVC App) to front-end the uploading of image file and presentation of the component.

Azure BLOB Storage (shown as one but can be multiple accounts) to upload the image, save the pre-processed image and extract the similar image result.

Azure Storage Queues (two of them) which enables messaging between the various components

Azure Redis Cache to track the progress of the query and hold the result as it moves from upload to pre-processing to query the ML Service.

Azure Functions – two of them – one of them that pre-processes the image, and the other that queries the ML Service with the resized image information and returns the result.

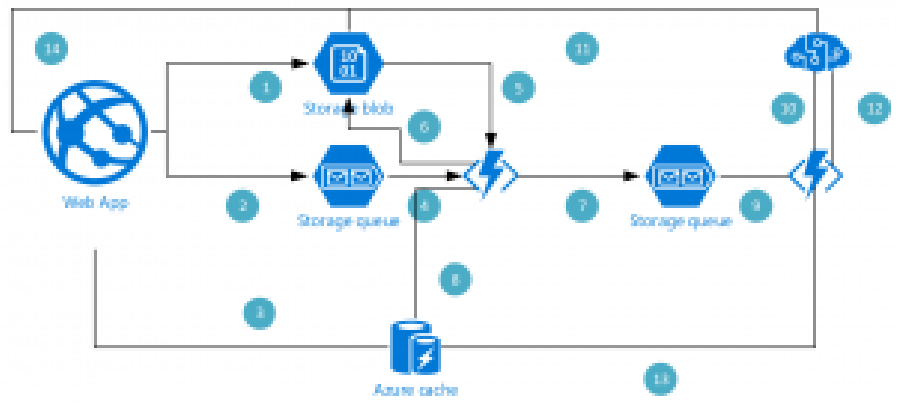

Figure: Architecture diagram

Data Flow

Web App uploads the image to storage blob.

Web App sends a message about the upload to the Azure Function through the queue with path and a tracking ID. (The tracking ID is generated by the web app and is the reason why we are not using the events from Azure Blob storage).

Web App puts an entry for the picture on the Redis cache and polls the cache to keep track of the update.

Azure Function reads the message of the queue.

Azure Functions reads the image from the blob storage and resizes it.

Azure Function writes the resized image into the blob storage.

Azure Function places a message on Storage Queue to call the ML service

Azure Function updates the status on the Redis cache.

Azure Function reads the message of the second queue.

Azure Function calls the ML Service with the path of the resized image and waits for response.

ML Services determines the closes match by reading the image off the BLOB storage

ML Service returns the result to the Azure Function

Azure Function updates the result on Azure Cache which is read by Web App in its next poll.

Web App reads the BLOB Storage to display images.

The figure below shows an example mobile app interface:

Figure: Example mobile app interface.

Please note:

The images shown above are just representative of the flow. The application flow at the time of writing this document is not connected to the ML service. This is intended to be an example. If the provided image is uploaded, resized and shared with a mock of the ML service to return synthetic results.