Automation with Taskoids: Case Study I

Taskoid TACuv1.0:Active Contour Segmentation of images to UV maps for the manufacture of Cuddles(™), a patented stuffed toy pillow with thermoregulation.

Dr Bheemaiah, Anil Kumar, A.B Seattle W.A 98125
miyawaki@yopmail.com

Abstract:
Taskoids are a quantification of task automation with IaC and MaC, in a SaaS model based on cloud computing(Bheemaiah, n.d.), the author presents a case study of active contour segmentation, using AWS Sagemaker with a tarball of a TensorFlow model. A case study of a genome for a pretrained model used with AWS Sage maker is presented, with extensions to training a network on AWS Sagemaker and also creating tensorflow models from Wolfram mathematical descriptions.("Website" n.d.)
A case study of the generation of the UV maps for manufacturing Cuddles(™) a stuffed toy with embedded Alexa controlled thermo-regulation, using the dog-ear framework and
Thermoregulation is cited as an alternative therapy in Autism Spectrum disorder. In this paper we design a Taskoid TACuv1.0 (Bheemaiah, n.d., n.d.)with Alexa enabled thermoregulation, for the design of Cuddlies(™) from images drawn by patients.
Keywords: Amazon Alexa, AWS, Sagemaker, active contour segmentation, resnet50, resnet100, deeplearning, Sonoff switches, Thermoregulation, Images, Cuddles(™), Autism Spectrum Disorder, climate change, carbon footprint.

What:
Design of TACuv1.0 , version 1.0 of a taskoid to convert a cartoon image to the UV maps needed for the manufacture of a Cuddles(™), for thermo-regulation, with or without conversational UI.
How:
The steps needed are:

1. Creation of an outline with a seam using contour segmentation, to determine the outline.
2. Persistence of the outline, without the cartoon image for a Cuddles(™).

Why:
Dog-ears, a formal system for the control and integration of wifi-enabled devices with C-UI or MUI, (Bheemaiah, n.d.)allows for the use of retro-switches, either active Sonoff switches or passive ones with templates or the use of IFTTT or the RAVA language. This enables passive control or active thermoregulation.

Applications:

Local body thermoregulation as a therapy, outdoor and backpacking use, energy savings in low-income households, lower cost of living, personal heating, lower carbon footprint, fighting climate change.

Summary:

Main Points:
Contour Segmentation using AWS Sagemaker or Tensor formulation using a pretrained network.
Deletion of cartoon and persistence of only the outline.

Code Base:

Introduction.

Problem Definition.

Design of TACuv1.0 , which includes a UI and a taskoid for IaC , a JSON for an AWS stack for transcription of the genome:
[model.tar.gz, [IAM],
Input_data.gz, SageMaker_Endpoint]

Background.

<original-contribution>

Formal Definitions:

For a given image I, implement a cloud function Active_Contour_Segmentation(I, [Im]) with a machine genome for a JSON or YAML representation to an AWS based C-UI or MUI.
The function can use Sagemakers, builtin ResNet or use a TensorFlow or PyTorch model tarball. A model can be trained on Sagemaker or a pre-trained model deployed. We consider a taskoid for a pre-trained model.
("Active Contour Model — Skimage v0.16.dev0 Docs" n.d.)
Given a wireframe W for the UI, we use the taskoids, TASSv1.0: for an automated cloud backend and visual UI, and the taskoid TACIv1.0: for a C-UI or MUI from W.
Green coding is often useful in the creation of cloud functions similar to API designs with queries, similar to the architecture of Bayou. In a companion paper, we define a query-based cloud function generator using a code search functionality.
We define queryCodeGenerator(Q, [f]).

ResNet for automatic contour segmentation.

Contour segmentation is amenable to both manual and Resnet based generation. A plethora of automated contour generation algorithms both procedural and using deep learning are available, the most prominent being the use of the ResNet architecture for contour generation.

AWS Sagemaker is directly amenable to active contour generation and a taskoid for this purpose is illustrated with a pre-trained ResNet RN, made available, the problem is then of a JSON representation for using a code snippet or TensorFlow model.
We use the documentation from AWS on deploying a tensorflow model, pre-trained with IAM permissions on a SageMaker Endpoint.
The genome is [model.tar.gz(yummy tar balls), [IAM],

Input_data.gz, SageMaker_Endpoint]

Code-Generators used for transcription are straight forward in CLI and python SDK code and the API's used.
An IAC JSON for the CLI and python scripting is readily generated as transcription.

Reproduced from, (aws n.d.)

We use a Docker container for TensorFlow, a similar transcription exists for PyTorch models.

Deploying a TensorFlow Serving Model

To use your TensorFlow Serving model on SageMaker, you first need to create a SageMaker Model. After creating a SageMaker Model, you can use it to create SageMaker Batch Transform Jobs for offline inference, or create SageMaker Endpoints for real-time inference.

Creating a SageMaker Model

A SageMaker Model contains references to a `model.tar.gz` file in S3 containing serialized model data, and a Docker image used to serve predictions with that model.

You must package the contents in a model directory (including models, inference.py and external modules) in .tar.gz format in a file named "model.tar.gz" and upload it to S3. If you're on a Unix-based operating system, you can create a "model.tar.gz" using the `tar` utility:

```
tar -czvf model.tar.gz 12345 code
```

where "12345" is your TensorFlow serving model version which contains your SavedModel.

After uploading your `model.tar.gz` to an S3 URI, such as `s3://your-bucket/your-models/model.tar.gz`, create a SageMaker Model which will be used to generate inferences. Set `PrimaryContainer.ModelDataUrl` to the S3 URI where you uploaded the `model.tar.gz`, and set `PrimaryContainer.Image` to an image following this format:

```
520713654638.dkr.ecr.{REGION}.amazonaws.com/sagemaker-tensorflow-serving:{TENSORFLOW_SERVING_VERSION}-{cpu|gpu}
```

For those using Elastic Inference set the image following this format instead:

```
520713654638.dkr.ecr.{REGION}.amazonaws.com/sagemaker-tensorflow-serving-eia:{TENSORFLOW_SERVING_VERSION}-cpu
```

Where `REGION` is your AWS region, such as "us-east-1" or "eu-west-1"; `TENSORFLOW_SERVING_VERSION` is one of the supported versions: "1.11" or "1.12"; and "gpu" for use on GPU-based instance types like ml.p3.2xlarge, or "cpu" for use on CPU-based instances like `ml.c5.xlarge`.

The code examples below show how to create a SageMaker Model from a `model.tar.gz` containing a TensorFlow Serving model using the AWS CLI (though you can use any language supported by the AWS SDK) and the SageMaker Python SDK.

AWS CLI

```
timestamp() {
  date +%Y-%m-%d-%H-%M-%S
}


MODEL_NAME="image-classification-tfs-$(timestamp)"
MODEL_DATA_URL="s3://my-sagemaker-bucket/model/model.tar.gz"

aws s3 cp model.tar.gz $MODEL_DATA_URL

REGION="us-west-2"
TFS_VERSION="1.12.0"
PROCESSOR_TYPE="gpu"
IMAGE="520713654638.dkr.ecr.$REGION.amazonaws.com/sagemaker-tensorflow-serving:$TFS_VERSION-$PROCESSOR_TYPE"

# See the following document for more on SageMaker Roles:
# https://docs.aws.amazon.com/sagemaker/latest/dg/sagemaker-roles.html
ROLE_ARN="[SageMaker-compatible IAM Role ARN]"

aws sagemaker create-model \
    --model-name $MODEL_NAME \
    --primary-container Image=$IMAGE,ModelDataUrl=$MODEL_DATA_URL \
    --execution-role-arn $ROLE_ARN
```

SageMaker Python SDK

```
import os
import sagemaker
from sagemaker.tensorflow.serving import Model

sagemaker_session = sagemaker.Session()
role = 'arn:aws:iam::038453126632:role/service-role/AmazonSageMaker-ExecutionRole-20180718T141171'
bucket = 'am-datasets'
prefix = 'sagemaker/high-throughput-tfs-batch-transform'
s3_path = 's3://{}/{}'.format(bucket, prefix)

model_data = sagemaker_session.upload_data('model.tar.gz',

bucket,

os.path.join(prefix, 'model'))
```

```
# The "Model" object doesn't create
a SageMaker Model until a Transform
Job or Endpoint is created.
tensorflow_serving_model =
Model(model_data=model_data,

role=role,

framework_version='1.13',

sagemaker_session=sagemaker_session)
```

After creating a SageMaker Model, you can refer to the model name to create Transform Jobs and Endpoints. Code examples are given below.

Creating a Batch Transform Job

A Batch Transform job runs an offline-inference job using your TensorFlow Serving model. Input data in S3 is converted to HTTP requests, and responses are saved to an output bucket in S3.

CLI

```
TRANSFORM_JOB_NAME="tfs-transform-jo
b"
TRANSFORM_S3_INPUT="s3://my-sagemake
r-input-bucket/sagemaker-transform-i
nput-data/"
TRANSFORM_S3_OUTPUT="s3://my-sagemak
er-output-bucket/sagemaker-transform
-output-data/"

TRANSFORM_INPUT_DATA_SOURCE={S3DataS
ource={S3DataType="S3Prefix",S3Uri=$
TRANSFORM_S3_INPUT}}
CONTENT_TYPE="application/x-image"
INSTANCE_TYPE="ml.p2.xlarge"
INSTANCE_COUNT=2

MAX_PAYLOAD_IN_MB=1
MAX_CONCURRENT_TRANSFORMS=16

aws sagemaker create-transform-job \
    --model-name $MODEL_NAME \
    --transform-input
DataSource=$TRANSFORM_INPUT_DATA_SOU
RCE,ContentType=$CONTENT_TYPE \
    --transform-output
S3OutputPath=$TRANSFORM_S3_OUTPUT \
    --transform-resources
InstanceType=$INSTANCE_TYPE,Instance
Count=$INSTANCE_COUNT \
    --max-payload-in-mb
$MAX_PAYLOAD_IN_MB \
    --max-concurrent-transforms
$MAX_CONCURRENT_TRANSFORMS \
    --transform-job-name $JOB_NAME
```

SageMaker Python SDK

```
output_path =
's3://my-sagemaker-output-bucket/sag
emaker-transform-output-data/'
tensorflow_serving_transformer =
tensorflow_serving_model.transformer
(

framework_version = '1.12',

instance_count=2,

instance_type='ml.p2.xlarge',

max_concurrent_transforms=16,

max_payload=1,

output_path=output_path)

input_path =
's3://my-sagemaker-input-bucket/sage
maker-transform-input-data/'
```

```
tensorflow_serving_transformer.transform(input_path,
content_type='application/x-image')
```

Creating an Endpoint

A SageMaker Endpoint hosts your TensorFlow Serving model for real-time inference. The InvokeEndpoint API is used to send data for predictions to your TensorFlow Serving model.

AWS CLI

```
ENDPOINT_CONFIG_NAME="my-endpoint-config"
VARIANT_NAME="TFS"
INITIAL_INSTANCE_COUNT=1
INSTANCE_TYPE="ml.p2.xlarge"
aws sagemaker create-endpoint-config \
    --endpoint-config-name $ENDPOINT_CONFIG_NAME \
    --production-variants VariantName=$VARIANT_NAME,ModelName=$MODEL_NAME,InitialInstanceCount=$INITIAL_INSTANCE_COUNT,InstanceType=$INSTANCE_TYPE

ENDPOINT_NAME="my-tfs-endpoint"
aws sagemaker create-endpoint \
    --endpoint-name $ENDPOINT_NAME \
    --endpoint-config-name $ENDPOINT_CONFIG_NAME

BODY="fileb://myfile.jpeg"
CONTENT_TYPE='application/x-image'
OUTFILE="response.json"
aws sagemaker-runtime invoke-endpoint \
    --endpoint-name $ENDPOINT_NAME \
    --content-type=$CONTENT_TYPE \
    --body $BODY \
    $OUTFILE
```

SageMaker Python SDK

```
predictor = tensorflow_serving_model.deploy(initial_instance_count=1,

framework_version='1.12',

instance_type='ml.p2.xlarge')
prediction = predictor.predict(data)
```

Enabling Batching

You can configure SageMaker TensorFlow Serving Container to batch multiple records together before performing an inference. This uses TensorFlow Serving's underlying batching feature.

You may be able to significantly improve throughput, especially on GPU instances, by enabling and configuring batching. To get the best performance, it may be necessary to tune batching parameters, especially the batch size and batch timeout, to your model, input data, and instance type.

You can set the following environment variables on a SageMaker Model or Transform Job to enable and configure batching:

```
# Configures whether to enable record batching.
# Defaults to false.
SAGEMAKER_TFS_ENABLE_BATCHING="true"

# Configures how many records
```

```
# Corresponds to "max_batch_size" in TensorFlow Serving.
# Defaults to 8.
SAGEMAKER_TFS_MAX_BATCH_SIZE="32"

# Configures how long to wait for a full batch, in microseconds.
# Corresponds to "batch_timeout_micros" in TensorFlow Serving.
# Defaults to 1000 (1ms).
SAGEMAKER_TFS_BATCH_TIMEOUT_MICROS="100000"

# Configures how many batches to process concurrently.
# Corresponds to "num_batch_threads" in TensorFlow Serving
# Defaults to number of CPUs.
SAGEMAKER_TFS_NUM_BATCH_THREADS="16"

# Configures number of batches that can be enqueued.
# Corresponds to "max_enqueued_batches" in TensorFlow Serving.
# Defaults to number of CPUs for real-time inference,
# or arbitrarily large for batch transform (because batch transform).
SAGEMAKER_TFS_MAX_ENQUEUED_BATCHES="10000"
```

IaC is in a JSON format from a CloudFormation Template.

`We create a template for Sagemaker from the needed CLI JSON, let the CLI be [J], then we use the` **`create-cloud-formation-template`**

command.(“Create-Cloud-Formation-Template — AWS CLI 1.16.240 Command Reference” n.d.) template.

The rest is a straightforward creation of a stack for the application by code generation from the template as a JSON. We leave this as an exercise for the reader.

Here are some Hints,

Creating a Stack

(“Creating a Stack - AWS CloudFormation” n.d.)

“To create a stack you run the `aws cloudformation create-stack` command. You must provide the stack name, the location of a valid template, and any input parameters.
Parameters are separated with a space and the key names are case sensitive. If you mistype a parameter key name when you run `aws cloudformation create-stack`, AWS CloudFormation doesn't create the stack and reports that the template doesn't contain that parameter.”

</original-contribution>

Discussion.

We have thus described the design of a parameter based genome and an A.I based transcription of the genome using code-generation modules. In the above case, CloudFormation Templates, provide a straightforward way to describe an AWS Stack as IaC, with the template, which can readily be created from CLI code, as JSON, integrating all needed API's. API's can easily be generated in Green Coding, described in an accompanying publication.Much more coding goes into code generator modules for logs and exceptions on the Sagemaker Endpoint, for more efficient TensorFlow tarball deployment.

Future Work.

In future work, we describe the taskoids from a mathematical tensor formulation of a tensorflow model to the generation of the model tarball, as a pretrained network, given the training set for the model, we can also write SageMaker Taskoids to train a tensorflow model.

References.

"Active Contour Model — Skimage v0.16.dev0 Docs." n.d. Accessed September 4, 2019. https://scikit-image.org/docs/dev/auto_examples/edges/plot_active_contours.html.

aws. n.d. "Aws/sagemaker-Tensorflow-Serving-Container." GitHub. Accessed September 18, 2019. https://github.com/aws/sagemaker-tensorflow-serving-container.

Bheemaiah, Anil Kumar. n.d. "Formal Methods for Multi Modal UI for Robotics." https://doi.org/10.31224/osf.io/7zdu3.

———. n.d. "Taskoids: A Formal Definition." https://doi.org/10.31224/osf.io/yhsbt.

"Create-Cloud-Formation-Template — AWS CLI 1.16.240 Command Reference." n.d. Accessed September 18, 2019. https://docs.aws.amazon.com/cli/latest/reference/serverlessrepo/create-cloud-formation-template.html.

"Creating a Stack - AWS CloudFormation." n.d. Accessed September 18, 2019. https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/using-cfn-cli-creating-stack.html.

"Website." n.d. Accessed September 22, 2019. https://resources.wolframcloud.com › NeuralNetRepository.