**CHAPTER 12:** Classifying Images using Amazon SageMaker

- 24 pages

### 

**Introduction**

Image classification has been one of the leading research fields in the last five years. It is not surprising because being able to successfully classify images solves many business problems across a variety of industries. For example, the entire autonomous vehicles industry is dependent on the accuracy produced by these image classification and object detection models.

One of the key challenges in classifying images is availability of large training datasets. For example, to create Amazon Go type experiences, the e-commerce retailer may have trained their machine learning algorithms on large volumes of images. When we do not have images covering all types of real world scenarios – scenarios ranging from time of the day (brightness), ambience around target item, and item angle – we’re unable to train image classification algorithms that are able to perform well in real-life environments. Furthermore, it takes a lot of effort to build convolutional neural network architecture that is optimal for the dataset at hand. The considerations range from number of convolutional layers to batch size to optimizer to drop out rates. It takes multiple trial and error experiments to arrive at an optimal model iteration.

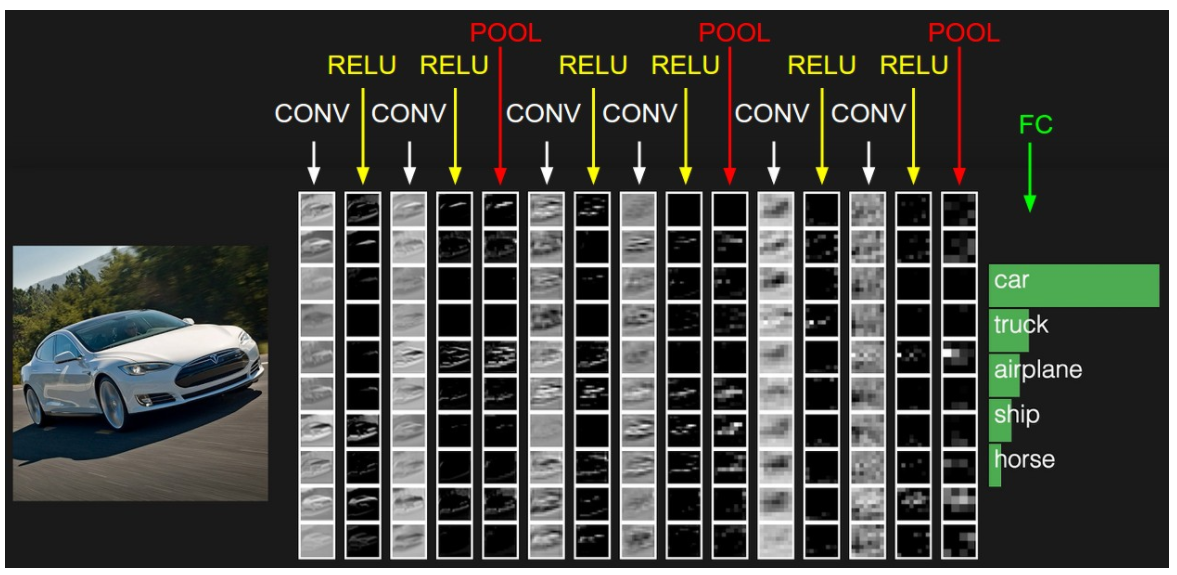
In this chapter, we will cover:

* An overview of Convolutional Neural Networks and Residual Networks
* Significance of Transfer Learning
* Image Classification through Transfer Learning
* Conduct inference on images through Batch Transform in SageMaker

Amazon SageMaker drastically simplifies the image classification problem. Aside from gathering rich set of images for training, all you will need to do is specify hyperparameters (parameter internal to the algorithm), training docker image, and infrastructure specifications for training.

The SageMaker Image Classification algorithm is an implementation of ResNets (Residual Networks). Before we deep dive into the details of the algorithm, let’s briefly understand convolutional neural networks and residual networks and how they learn patterns in images.

**Convolutional Neural Networks**, like an any other neural network, are made up of input, hidden and output layers, with learnable weights and biases. These learned parameters produce an optimal network that can, for example, classify images. The weights and biases can be adjusted through an appropriate optimizer, such as Stochastic Gradient Descent (SGD), with back propagation. However, the difference between any feedforward artificial neural network and CNNs is that the hidden layers in CNNs are convolutional layers. Each convolutional layer consists of one or more filters. The job of these filters is to recognize patterns in input images. These filters can have varying shapes, ranging from 1x1 to 3x3 and so on, and are initialized with random weights. As the input image passes through the convolutional layer, each filter will slide over every 3x3 block of pixels (in the case of 3x3 filter) until the entire image is covered. This sliding is referred to as convolving. During the process of convolving, dot product is applied to filter weights and pixel values in the 3x3 block, thus learning image features. The initial layers of CNN learn basic geometric shapes, such as edges, circles. While the later layers learn more sophisticated objects, such as eyes, ears, feathers, beaks, full cats & dogs.

For example, in the ConvNet below that is being trained on car image, the initial layer of the network contains raw image pixels, while the last fully connected (FC) layer contains scores for each class (here we show just 5 classes). The conv layer transforms input image through weights and biases, while RELU layer transforms its input by producing values that are >= 0 (max(0,x)). The POOL layer down samples its input, going from 32x32x12 to 16x16x12, for instance, while FC layer computes class score.

*Source: Andrej Karpathy Stanford University (cs231): http://cs231n.github.io/convolutional-networks/#overview*

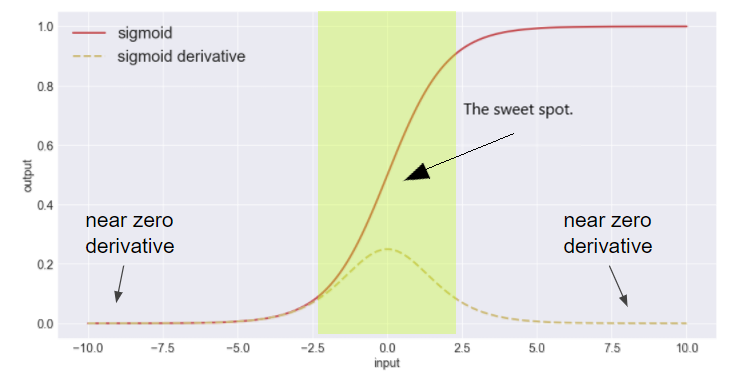
**Residual Networks and Transfer learning**

**Residual Networks**. Now that we know how CNNs work, it is time to unpack *residual networks*

Why do we need residual networks?

*Vanishing / exploding gradients*: With deeper convolutional neural networks, as we stack more layers to learn complex features, vanishing/exploding gradients issues result. In other words, during the training process, some neurons die (do not activate), causing vanishing gradient problem. This happens when activation function receives input with varying distributions (for example, if you’re passing black and white images of cats vs colored ones through the network trained on colored cats, the input raw pixels belong to a different distribution, causing vanishing gradient problem) . If we restrict neuron output to area around zero, we can ensure that each layer will pass a substantive gradient back to prior layers. Exploding gradients, on the other hand, occur when large error gradients accumulate, resulting in very large updates to neural network models. These large error gradients occur when gradients that have values > 1 are multiplied across network layers.

The following figure shows that sigmoid activation will pass a measurable gradient (sigmoid derivative) back to prior layers when input is restricted/normalized to area around zero.



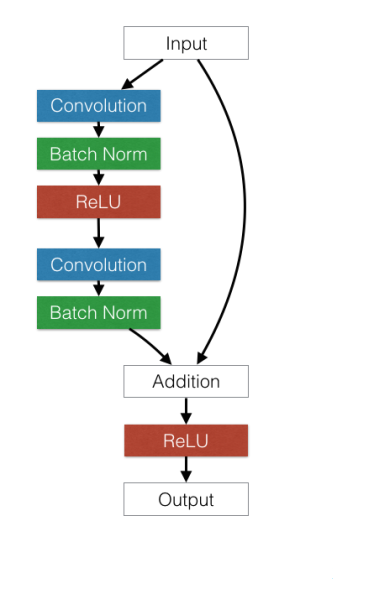
*Source: Sigmoid Activation Function: https://towardsdatascience.com/intuit-and-implement-batch-normalization-c05480333c5b*

To some degree, this problem can be addressed through batch normalization layers, where the output of previous layer is normalized by subtracting batch mean (shifting) and dividing by the batch standard deviation (scaling).

*What is batch in batch normalization?* As the deep neural networks are trained, the entire training dataset is divided into small batches. These batches of training samples are used to calculate model error and update model weights.

*Harder Optimization:* An additional problem that with conv nets is *degradation* in training accuracy as the network gets deeper – not all systems/data sets are easy to optimize. When the network gets deeper, the number of parameters to optimize becomes large, resulting in a tough to train network. Adding more layers results in even more training errors – difficult to converge. Deep residual learning comes to the rescue, which combines learnings from previous layers (x) with learnings from shallower model (F(x) – convolutional layer) – F(x) + x.

**ResNet Building Block**

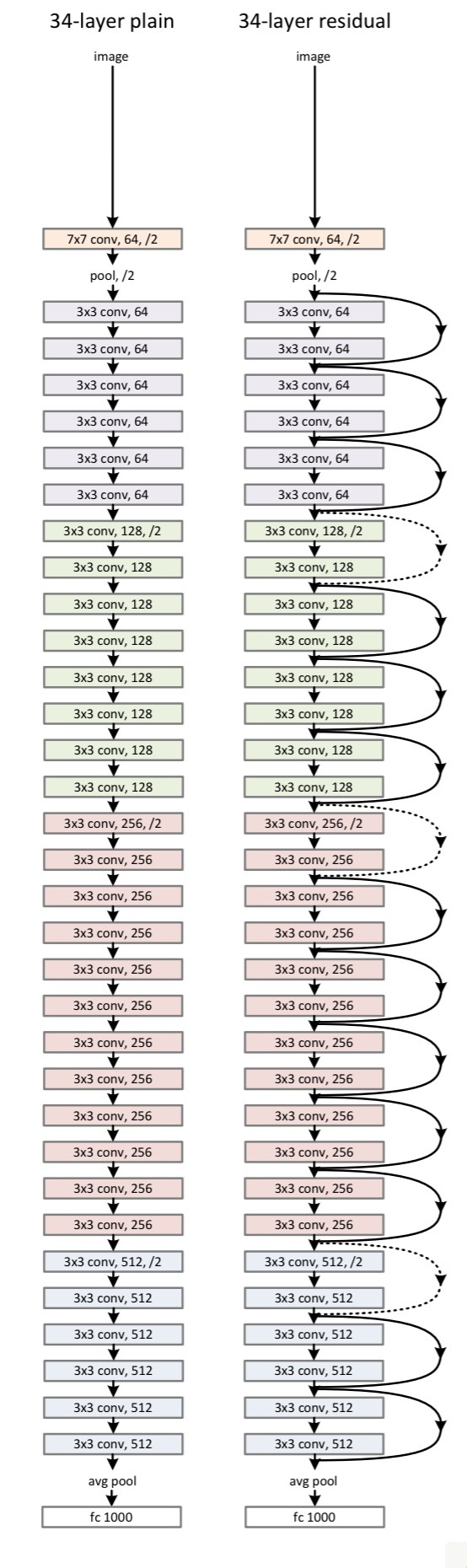


*Source: Study of Residual Networks for Image Recognition - Stanford University*

In Residual Networks (ResNet), after the input image is convolved for feature extraction, the feature map is batch normalized for increased stability. The Rectified Linear Unit (ReLU) activation function is used to make ResNets non-linear classification models. The building block above shows two convo + batch norm layers stacked on top of each other. These layers fit a residual mapping. We can represent original mapping as H(x) – a.k.a Hypothesis. Therefore, the residual mapping F(x) can represented as H(x) – x; i.e. residual mapping can only represent part of the original hypothesis. It is easy to optimize residual mapping than to optimize the original mapping. F(x) + x can be represented using a feed forward neural network with “shortcut connections”. Shortcut connections skip one or more layers. In our case, these connections conduct identity mapping. And their outputs are added to outputs of stacked layers. The entire network can still be optimized by Stochastic Gradient Descent (SGD) with back propagation. As you can see from the building block above, Residual Networks are not computationally complex when compared to CNNs. However, they offer much better accuracy and stability than CNNs.

Compared to convolutional layers, the shortcut connections are always alive, allowing the gradients to easily pass through them.

The following picture is taken from He et. al. 2015 paper – It clearly compares Convolutional Neural Nets and Residual Nets. As you can see, Residual Nets have skip connections that pass on the learnings from previous layers. And these skip connections run parallel to convolutional layers that are used for residual mapping.



*Source: He et al., 2015. Deep residual networks for image recognition*

**Transfer Learning.** Because image classification requires a large volume of images for training convolutional networks, an alternative approach can be used to classify images when size of training dataset is small. Transfer learning enables you apply the knowledge of an already trained model to a different but related problem. We can reuse weights of pre-trained deep learning model trained on millions of images and fine-tune the network with new/custom dataset unique to your business case. Through transfer learning, low level geometric features, such as edges, can already be recognized by pre-trained ResNet-18 (18 layer network). However, for mid to high-level feature learning, the top fully connected (FC) layer (refer to the picture above) is reinitialized with random weights. Then the whole network is fine-tuned with the new data – the random weights are adjusted by passing training data through the network and using an optimization technique -- for example, stochastic gradient descent with back propagation.

In the below example, we will use ResNet-18 trained on ImageNet dataset. ImageNet is one of the largest datasets used in the image and vision research fields. It has more than 11 million images belonging to around 11,000 categories.

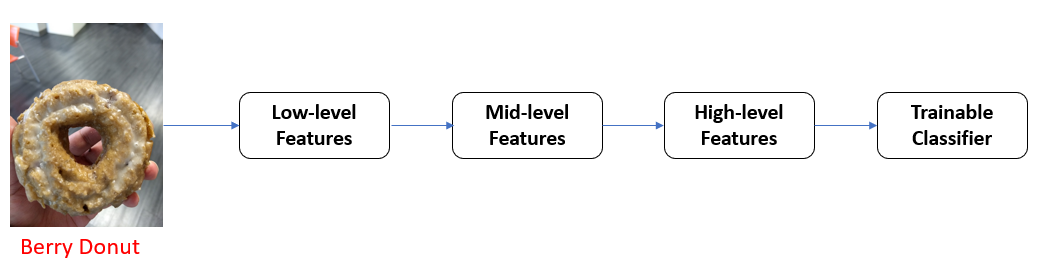
**Image Classification through Transfer Learning in Amazon SageMaker**

In this section, we’ll employ SageMaker’s Image Classification algorithm in transfer learning mode to classify some bakery and fast food items.

The Image Classification algorithm implemented Residual Networks (ResNets) to categorize images. We can either train ResNets from scratch or use pre-trained networks. Since we have small image dataset to train, we’ll use 18-layer pre-trained ResNet trained on ImageNet dataset.

Our new dataset contains around 302 images with 5 categories (Hot Dog, Berry Donut, Glazed Twist, Muffin, and Peanut Butter Cookie). Each of the items contain 40 to 90 images, covering the item from varied angles, brightness, contrast, and size.

The Image Classifier learns low level features from pre-trained ResNet and high-level features by training the same ResNet-18 with new dataset



To get started, we need to set up the environment with a few prerequisite steps -- permissions, and configurations.

**Preprocessing**

*Permissions and setup*

Here, we set up authentication to AWS services required for training and define variables required throughout the notebook. There are three parts to this:

\* The role used to give SageMaker training and hosting services access to input data. This is the same role used to start the notebook

\* The S3 bucket that you want to use for training and to save model data

\* The Amazon Sagemaker image classification docker image

role = get\_execution\_role()

bucket = 'ai-in-aws'

training\_image = get\_image\_uri(boto3.Session().region\_name, 'image-classification')

print(training\_image)

*Convert Train/Val Datasets to RecordIO format*

Amazon SageMaker Image Classification algorithm accepts images in file mode via two content types -- namely RecordIO (application/x-recordio) and image (image/png, image/jpeg, and application/x-image). In this notebook, we will use RecordIO format.

RecordIO is a binary format for representing images efficiently. The format enables --

\* Storing images in a compact format (reduces the size of dataset)

\* Packing data together allows continuous reading on the disk

\* Partitioning data, simplifying distributed setting

The training and validation images are available in zipped format on s3 bucket. In the section below, you will unzip the training and validation images and create corresponding list files. Each of the training and validation folders contain subfolder for every image category. As you will see below, the root folder (./merch\_data) has subfolder (./merch\_data/Hot\_Dog\_1) for every category

List files are used to summarize images across multiple classifications/categories. For example, list files have the following format,

<Image index> <image label/category> <image path>

225 3.000000 Hot\_Dog\_1/IMG\_20180711\_180420838.jpg

From the list files, we then create compressed representation of images via RecordIO format. MXNet offers im2rec.py module to perform this conversion.

Let’s begin with defining item categories and uploading test images to designated S3 bucket

s3\_prefix = 'image-classification-merchandise'

s3\_train\_fname = 'Train.zip' # name of training dataset

s3\_val\_fname = 'Validation.zip' # name of validation dataset

s3\_client = boto3.client('s3')

item\_categories = ['Berry\_Donut', 'Glazed\_Twist', 'Hot\_Dog\_1', 'Muffin', 'Peanut\_Butter\_Cookie']

# 4.000000 Peanut\_Butter\_Cookie

# 2.000000 Hot\_Dog\_1

# 3.000000 Muffin

# 0.000000 Berry\_Donut

# 1.000000 Glazed\_Twist

# Upload test data to s3 bucket

batch\_input = 's3://{}/image-classification-merchandise/test/'.format(bucket)

test\_images = 'merch\_data/test/'

!aws s3 cp $test\_images $batch\_input --recursive --quiet

*User defined functions*

We will now define utility functions that will be used throughout the notebook.

run UtilityFunctions.py

*Upload\_to\_s3* function enables you to upload files -- in this case recordio representations of training and validation datasets – to S3 bucket

# Upload compact representation of images (RecordIO) to S3 bucket

def upload\_to\_s3(bucket, channel, file):

"""

input: S3 bucket name, folder on the bucket, RecordIO file to upload

"""

s3 = boto3.resource('s3')

data = open(file, "rb") # read in binary mode

key = channel + '/' + file

#Key is location on S3 bucket, Body - binary data

s3.Bucket(bucket).put\_object(Key=key, Body=data)

*Extract\_zipfile* allows you to download zipped training and validation datasets from S3 bucket to the SageMaker notebook instance.

# Read zipped images folder present on S3 bucket

def extract\_zipfile(bucket, key, rel\_path):

"""

input: S3 bucket name, location of zip file on s3 bucket, extract to location

"""

s3 = boto3.resource('s3')

obj = s3.Bucket(bucket).Object(key)

with io.BytesIO(obj.get()["Body"].read()) as bf:

bf.seek(0)

# Read the file as zipfile

with ZipFile(bf, mode='r') as zipf:

zipf.extractall(path=os.path.join('.', rel\_path))

*Create\_listfile* is used to iterate through training and validation image folders and create list files. MxNet’s im2rec (image to recordio) module is used both to create list and recordio files.

# Create List file for all images present in a directory

def create\_listfile(data\_path, prefix\_path):

"""

input: location of data, list file name and path

"""

# Obtain the path of im2rec.py on the current ec2 instance

im2rec\_path = mx.test\_utils.get\_im2rec\_path()

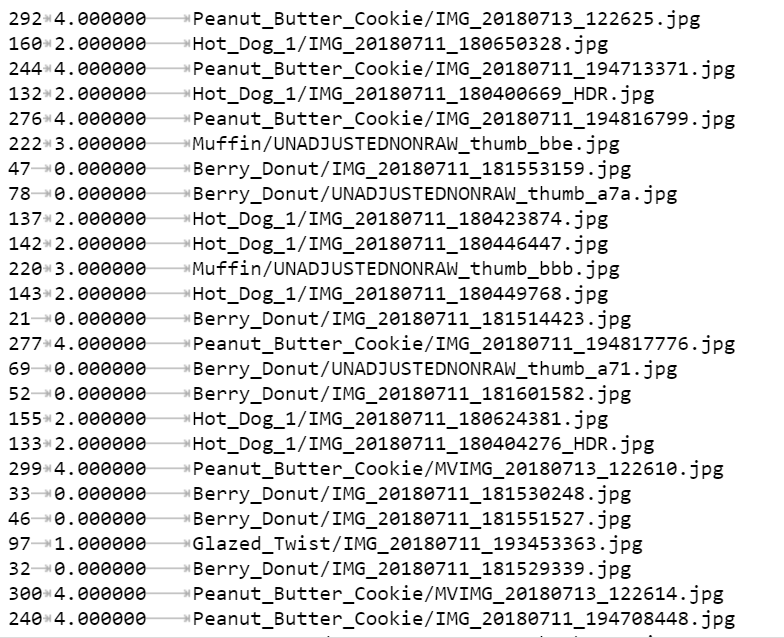
#Do not print output from the shell command - python im2rec.py --list --test-ratio=0.2 'merch\_data/merch-train.lst' 'merch\_data/train'

with open(os.devnull, 'wb') as devnull:

subprocess.check\_call(['python', im2rec\_path, '--list', '--recursive', prefix\_path, data\_path],

stdout=devnull)

*Excerpt of training list file*



*Create\_recordio* is used to create compact representation of training and validation datasets. RecordIO is the preferred format to send training data to SageMaker’s Image Classification algorithm.

# Create compact representation of images (RecordIO)

def create\_recordio(data\_path, prefix\_path):

"""

input: location of data, list file name and path

"""

# Obtain the path of im2rec.py on the current ec2 instance

im2rec\_path = mx.test\_utils.get\_im2rec\_path()

with open(os.devnull, 'wb') as devnull:

subprocess.check\_call(['python', im2rec\_path, '--num-thread=4', '--pass-through', prefix\_path, data\_path],

stdout=devnull)

*get\_items* is used to list objects in an S3 bucket. We will use this function when iterating over images in *test* folder at the time inference

# List objects from designated folder in S3 bucket

def get\_items(s3\_client, bucket, prefix):

"""

input: S3 client, S3 bucket name, folder on S3 bucket

output: names of images in the folder

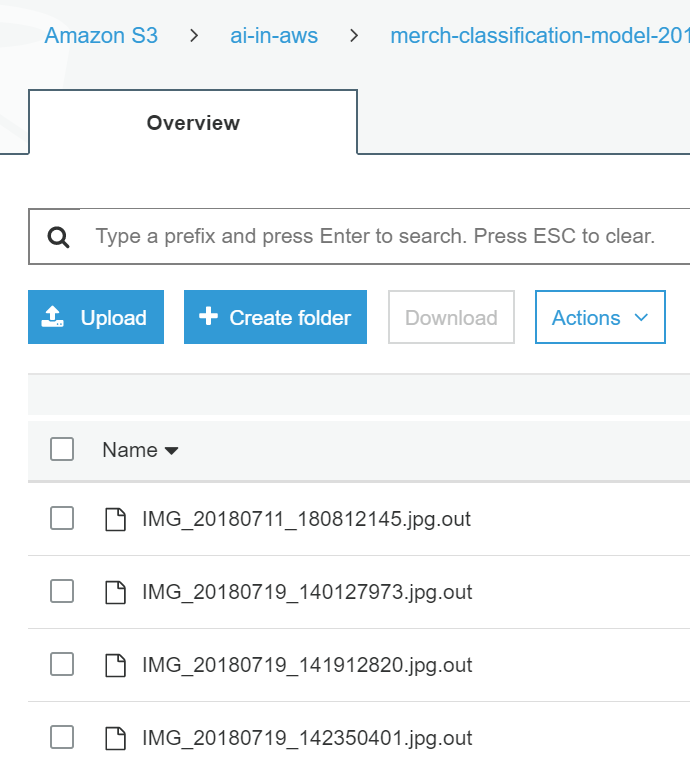
"""

response = s3\_client.list\_objects(Bucket=bucket, Prefix=prefix)

items = [content['Key'] for content in response['Contents']]

return items

*Get\_label\_image* is used to iterate through the folder containing inference output produced by trained Image Classification algorithm. For a given image name, the goal is to find the category with highest probability.



# Get label for each of the items/objects in output folder

def get\_label\_img(s3\_client, bucket, prefix, test\_images, item\_categories):

"""

input: S3 client, S3 bucket name, folder on S3 bucket containing output from image classification,

local path for test dataset, array of labels for each of the categories

output: predicted label and probability

"""

filename = prefix.split('/')[-1]

s3\_client.download\_file(bucket, prefix, filename)

with open(filename) as f:

data = json.load(f)

index = np.argmax(data['prediction']) #returns position of largest value

probability = data['prediction'][index] # get the entry with highest probability

# Format of predictions: {"prediction": [0.23142901062965393, 0.04651607573032379, 0.5318375825881958, 0.1708219051361084, 0.019395463168621063]}

print("Result: label - " + item\_categories[index] + ", probability - " + str(probability))

#imdecode is used to load raw image files

img = mx.image.imdecode(open(os.path.join(test\_images, os.path.splitext(filename)[0]), 'rb').read())

plt.imshow(img.asnumpy()); plt.show()

return item\_categories[index], probability

*Initialize environment variables*

Define the location where list files need to be saved on local SageMaker notebook instance

# Initialize variables

train\_key = os.path.join(s3\_prefix, s3\_train\_fname)

val\_key = os.path.join(s3\_prefix, s3\_val\_fname)

path = 'merch\_data'

train\_prefix = 'train'

val\_prefix = 'val'

# Define list file names

listfile\_train\_prefix = os.path.join(path,'merch-train')

listfile\_val\_prefix = os.path.join(path,'merch-val')

# Define location where the list files need to be saved

rel\_train\_path = os.path.join(path, train\_prefix)

rel\_val\_path = os.path.join(path, val\_prefix)

*Create and Upload RecordIO files to S3*

We will now take the following steps to create RecordIO files for training and validation datasets.

Step 1: Extract zip files, both training and validation

Step 2: Create list files for training and validation

Step 3: Create Record IO files for training and validation

# Extract training and validation zipped folders to merch\_data/<train/val>

extract\_zipfile(bucket, train\_key, rel\_train\_path)

extract\_zipfile(bucket, val\_key, rel\_val\_path)

# Create List files (./merch\_data)

create\_listfile(rel\_train\_path, listfile\_train\_prefix) #data path, prefix path

create\_listfile(rel\_val\_path, listfile\_val\_prefix)

# # Create RecordIO file

# data path --> prefix path (location of list file)

# mxnet's im2rec.py uses ./merch\_data folder to locate .lst files for train and val

# mxnet's im2rec.py uses ./merch\_data/<train/val> as data path

# list files are used to create recordio files

create\_recordio(rel\_train\_path, listfile\_train\_prefix)

create\_recordio(rel\_val\_path, listfile\_val\_prefix)

# Move Record IO file to the same location as the notebook

# destination, source

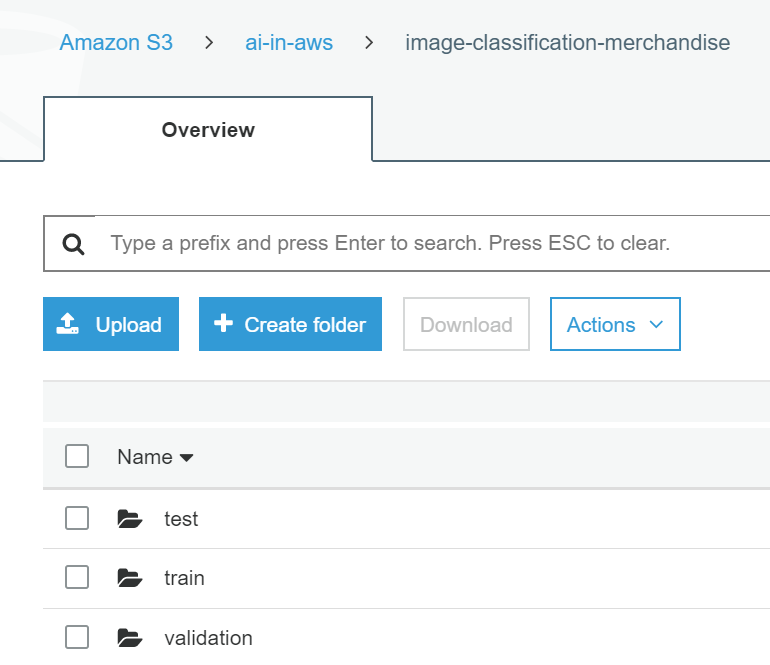
!rm 'merch-train.rec'

!rm 'merch-val.rec'

shutil.move(os.path.join(path, 'merch-train.rec'), '.')

shutil.move(os.path.join(path, 'merch-val.rec'), '.')

Upload RecordIO files to the train and validation folders on designated S3 bucket



s3\_train\_key = 'image-classification-merchandise/train'

s3\_validation\_key = 'image-classification-merchandise/validation'

s3\_train = 's3://{}/{}'.format(bucket, s3\_train\_key)

s3\_validation = 's3://{}/{}'.format(bucket, s3\_validation\_key)

upload\_to\_s3(bucket, s3\_train\_key, 'merch-train.rec')

upload\_to\_s3(bucket, s3\_validation\_key, 'merch-val.rec')

**Training**

We are now ready to train the image classification algorithm. There are two kinds of parameters that we specify:

1. Parameters for the training job. The input and output configuration, including the type of infrastructure to provision
2. Hyperparameters that are specific to algorithm

* num\_layers: The number of layers for the network. In this example, we will use the default 18 layers
* image\_shape: Since we are using pre-trained ResNet-18, we will keep our image size consistent with the specifications of ImageNet dataset
* num\_training\_samples: This is the total number training data points. In our case, it is set to 302
* num\_classes: This is number of categories. For our dataset, it is 5. We will classify 5 merchandise
* mini\_batch\_size: The number of training samples used for each mini batch. In a single machine multi-GPU setting, each GPU handles mini\_batch\_size/num of GPUs samples. In the case of distributed training, where multiple machines are involved, the actual batch size is number of machines \* mini\_batch\_size
* epochs: Number of iterations to go through to train the classification algorithm
* learning\_rate: Define how big the steps should be when back propagating to reduce loss. In the case of transfer learning, we will take smaller steps, so we can incrementally train the pre-trained network

*Define Hyperparameters*

# The algorithm supports multiple network depth (number of layers). They are 18, 34, 50, 101, 152 and 200

# For this training, we will use 18 layers

num\_layers = 18

image\_shape = "3,224,224" # Number of channels for color image, Number of rows, and columns (blue, green and red)

num\_training\_samples = 302 # number of training samples in the training set

num\_classes = 5 # specify the number of output classes

mini\_batch\_size = 60 # batch size for training

# number of epochs

epochs = 4

# learning rate

learning\_rate = 0.01

top\_k=2

# Since we are using transfer learning, we set use\_pretrained\_model to 1 so that weights can be

# initialized with pre-trained weights

use\_pretrained\_model = 1

#augmentation\_type = 'crop\_color\_transform'

*Specify Training Job Configuration*

Let’s begin by setting algorithm specifications of training job – image classification docker image; training input mode (File vs Pipe mode. Pipe mode is a recent addition to SageMaker toolkit, where input data is fed on the fly to algorithm container, without downloading it before training)

%%time

s3 = boto3.client('s3')

# create unique job name

job\_name\_prefix = 'merch-imageclassification'

timestamp = time.strftime('-%Y-%m-%d-%H-%M-%S', time.gmtime())

job\_name = job\_name\_prefix + timestamp

training\_params = \

{

# specify the training docker image

"AlgorithmSpecification": {

"TrainingImage": training\_image,

"TrainingInputMode": "File"

},

Define the location of training output (S3OutputPath), along with number and type of EC2 instances to provision and hyperparameters.

"RoleArn": role,

"OutputDataConfig": { # location of output

"S3OutputPath": 's3://{}/{}/output'.format(bucket, job\_name\_prefix)

},

"ResourceConfig": { # provision one GPU machine

"InstanceCount": 1,

"InstanceType": "ml.p2.xlarge",

"VolumeSizeInGB": 50

},

"TrainingJobName": job\_name,

"HyperParameters": {

"image\_shape": image\_shape,

"num\_layers": str(num\_layers),

"num\_training\_samples": str(num\_training\_samples),

"num\_classes": str(num\_classes),

"mini\_batch\_size": str(mini\_batch\_size),

"epochs": str(epochs),

"learning\_rate": str(learning\_rate),

"use\_pretrained\_model": str(use\_pretrained\_model)

# "augmentation\_type": str(augmentation\_type)

},

Specify *train* and *validation* channels: location of training and validation data. As for distribution training, the algorithm currently only supports *fullyreplicated* mode, where data is copied onto each machine

"StoppingCondition": {

"MaxRuntimeInSeconds": 360000

},

"InputDataConfig": [

{

"ChannelName": "train",

"DataSource": {

"S3DataSource": {

"S3DataType": "S3Prefix",

"S3Uri": s3\_train,

"S3DataDistributionType": "FullyReplicated"

}

},

"ContentType": "application/x-recordio",

"CompressionType": "None"

},

{

"ChannelName": "validation",

"DataSource": {

"S3DataSource": {

"S3DataType": "S3Prefix",

"S3Uri": s3\_validation,

"S3DataDistributionType": "FullyReplicated"

}

},

"ContentType": "application/x-recordio",

"CompressionType": "None"

}

]

}

#Training data should be inside a subdirectory called "train"

#Validation data should be inside a subdirectory called "validation"

#The algorithm currently only supports fullyreplicated model (where data is copied onto each machine)

"InputDataConfig": [

{

"ChannelName": "train",

"DataSource": {

"S3DataSource": {

"S3DataType": "S3Prefix",

"S3Uri": s3\_train,

"S3DataDistributionType": "FullyReplicated"

}

},

"ContentType": "application/x-recordio",

"CompressionType": "None"

},

{

"ChannelName": "validation",

print('Training job name: {}'.format(job\_name))

print('\nInput Data Location: {}'.format(training\_params['InputDataConfig'][0]['DataSource']['S3DataSource']))

Training job name: merch-imageclassification-2019-03-13-11-40-32

Input Data Location: {'S3DataType': 'S3Prefix', 'S3Uri': 's3://ai-in-aws/image-classification-merchandise/train', 'S3DataDistributionType': 'FullyReplicated'}

*Run the Training Job*

We will provide *training parameters* defined above as input to *create\_training\_job* method SageMaker client. Once the training job is created, we check its status

# create the Amazon SageMaker training job

sagemaker = boto3.client(service\_name='sagemaker')

sagemaker.create\_training\_job(\*\*training\_params)

# confirm that the training job has started

status = sagemaker.describe\_training\_job(TrainingJobName=job\_name)['TrainingJobStatus']

print('Training job current status: {}'.format(status))

Training job current status: InProgress

Training job ended with status: Completed

*Plot Training Results*

We will now plot results to evaluate training and validation accuracy of ResNet-18. We want to ensure that we’ve not overfit the network – scenario where validation accuracy decreases as training accuracy increases.

The results from training are available from CloudWatch logs. Once we create *logs* client, we pass log group and stream names to get log events of interest.

log\_client = boto3.client('logs')

lgn = '/aws/sagemaker/TrainingJobs' # log group name

lsn = '<training-job-name>/algo-1-1552477444' #specify your log stream name

log = log\_client.get\_log\_events(logGroupName=lgn, logStreamName=lsn)

trn\_acc = []

val\_acc = []

# iterate through the events in log group name and log stream name

# Get train and validation accuracy lists

for e in log['events']:

msg = e['message']

#print(msg)

if 'Validation-accuracy' in msg:

val = msg.split('=')

if 'INFO' not in val[0]:

val = val[1]

val\_acc.append(float(val))

if 'Train-accuracy' in msg:

trn = msg.split('=')

if 'INFO' not in trn[0]:

trn = trn[1]

trn\_acc.append(float(trn))

print('Maximum validation accuracy: %f '%max(val\_acc))

fig, ax = plt.subplots()

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

trn\_plot = ax.plot(range(epochs), trn\_acc, label='Training accuracy')

val\_plot = ax.plot(range(epochs), val\_acc, label='Validation accuracy')

plt.show()

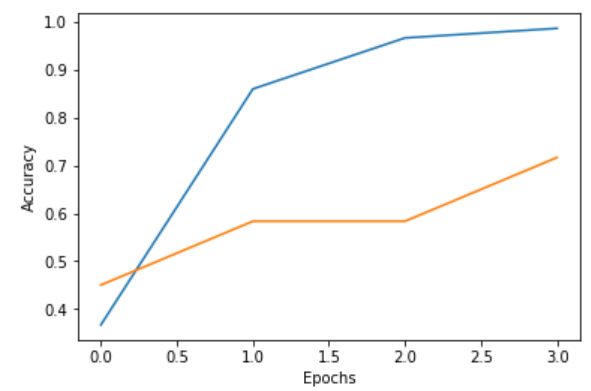
**trn\_acc**

[0.366667, 0.86, 0.966667, 0.986667]

**val\_acc**

[0.45, 0.583333, 0.583333, 0.716667]

Train = Blue color; Validation = Orange Color



As we can see from the graph above, trained ResNet model seems to have picked up enough patterns from fast food and bakery images. It is now time to deploy the trained model for inferences

**Perform Inferences through Batch Transform**

*Create the Model*

Let’s first create the model from the training job.

sage = boto3.Session().client(service\_name='sagemaker')

model\_name="MERCH-image-classification-model" + model\_type

print(model\_name)

info = sage.describe\_training\_job(TrainingJobName=job\_name)

Let’s retrieve model artifacts (location of trained model) from the completed training job. We will now use *create\_model*  method of SageMaker client to create model by passing model artifacts and image classification docker image.

info = sage.describe\_training\_job(TrainingJobName=job\_name)

# Get S3 location of the model artifacts

model\_data = info['ModelArtifacts']['S3ModelArtifacts']

print(model\_data)

# Get the docker image of image classification algorithm

hosting\_image = get\_image\_uri(boto3.Session().region\_name, 'image-classification')

primary\_container = {

'Image': hosting\_image,

'ModelDataUrl': model\_data,

}

# Create model

create\_model\_response = sage.create\_model(

ModelName = model\_name,

ExecutionRoleArn = role,

PrimaryContainer = primary\_container)

print(create\_model\_response['ModelArn'])

Why batch transform for prediction?  
### Create Batch Transform Job

s3://ai-in-aws/merch-imageclassification/output/merch-imageclassification-2019-03-13-11-40-32/output/model.tar.gz

*Create Batch Transform Job*

Since we want to classify more than 1 image at a time, we will create batch transform job to classify 4 images. Refer to the prior chapter on *Create Machine Learning Inference Pipelines with Amazon SageMaker* on when and where batch transform jobs are used and how they work.

%%time

timestamp = time.strftime('-%Y-%m-%d-%H-%M-%S', time.gmtime())

batch\_job\_name = "merch-classification-model" + timestamp

request = \

{

"TransformJobName": batch\_job\_name,

"ModelName": model\_name,

"MaxConcurrentTransforms": 16,

"MaxPayloadInMB": 6,

"BatchStrategy": "SingleRecord",

"TransformOutput": {

"S3OutputPath": 's3://{}/{}/output'.format(bucket, batch\_job\_name)

},

"TransformInput": {

"DataSource": {

"S3DataSource": {

"S3DataType": "S3Prefix",

"S3Uri": batch\_input

}

},

We will specify batch transform job configuration: job name, underlying trained model name, output location for batch transform job, location of test dataset (transform job input), transform job resources (type and number of EC2 instances to create).

"ContentType": "application/x-image",

"SplitType": "None",

"CompressionType": "None"

},

"TransformResources": {

"InstanceType": "ml.p2.xlarge",

"InstanceCount": 1

}

}

print('Transform job name: {}'.format(batch\_job\_name))

print('\nInput Data Location: {}'.format(batch\_input))

Pass the batch transform job configuration to *create\_transform\_job method* of SageMaker client (low-level client representing Amazon SageMaker service) , creating batch transform job.

sagemaker = boto3.client('sagemaker')

sagemaker.create\_transform\_job(\*\*request)

print("Created Transform job with name: ", batch\_job\_name)

while(True):

response = sagemaker.describe\_transform\_job(TransformJobName=batch\_job\_name)

status = response['TransformJobStatus']

if status == 'Completed':

print("Transform job ended with status: " + status)

break

if status == 'Failed':

message = response['FailureReason']

print('Transform failed with the following error: {}'.format(message))

raise Exception('Transform job failed')

time.sleep(30)

Created Transform job with name: merch-classification-model-2019-03-13-11-59-13

Transform job ended with status: Completed

*Review the results*

Navigate the batch transform output and test dataset folders on S3 bucket to review results. For each of the images in the test dataset, we will print their highest classification probability – i.e. what does the trained model classify an input image as.

inputs = get\_items(s3\_client, bucket, urlparse(batch\_input).path.lstrip('/'))

print("Sample inputs: " + str(inputs[:1]))

outputs = get\_items(s3\_client, bucket, batch\_job\_name + "/output")

print("Sample output: " + str(outputs[:1]))

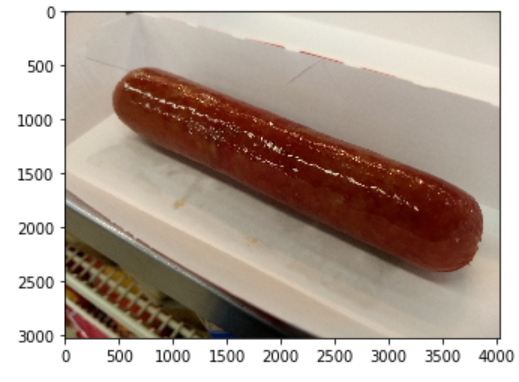
# Check prediction result of all the images

[get\_label\_img(s3\_client, bucket, prefix, test\_images, item\_categories) for prefix in outputs]

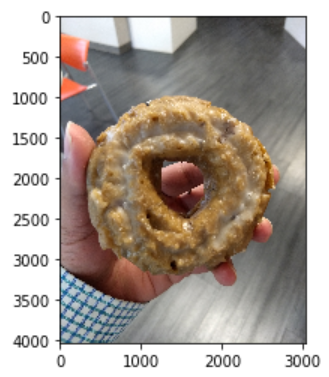
Sample inputs: ['image-classification-merchandise/test/IMG\_20180711\_180812145.jpg']

Sample output: ['merch-classification-model-2019-03-13-11-59-13/output/IMG\_20180711\_180812145.jpg.out']

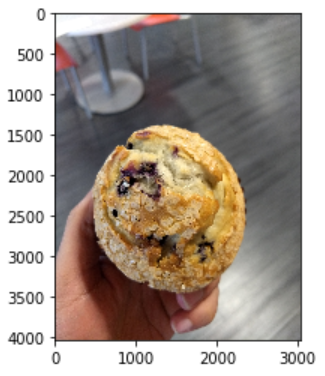
Result: label - Hot\_Dog\_1, probability - 0.9236235022544861



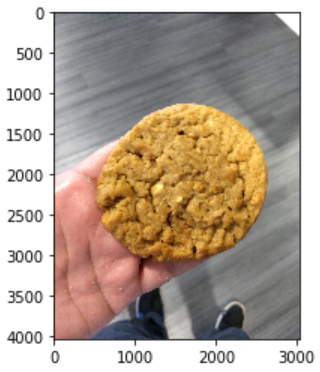
Result: label - Berry\_Donut, probability - 0.9964750409126282



Result: label - Muffin, probability - 0.6657814979553223



Result: label - Muffin, probability - 0.8914210200309753



[('Hot\_Dog\_1', 0.9236235022544861),

('Berry\_Donut', 0.9964750409126282),

('Muffin', 0.6657814979553223),

('Muffin', 0.8914210200309753)]

As we can see, out of the 4 images, 3 images are classified correctly. The 4th image, Peanut Butter Cookie, is incorrectly classified as Muffin. Although, the Peanut Butter Cookie in this image looks like the top of a muffin. The accuracy of the model can be improved through hyperparameter tuning and collecting large volumes of fast food and bakery images.

In this chapter, you’ve learned to seamlessly classify images using Amazon SageMaker’s Image Classification algorithm. You’ve learned to preprocess training and validation data; train ResNet-18 by specifying hyperparameters and infrastructure specifications; deploy trained model as batch transform job that can score test images.

**References**

*MXNet estimator in SageMaker*

<https://medium.com/devseed/use-label-maker-and-amazon-sagemaker-to-automatically-map-buildings-in-vietnam-a63090fb399f>

Vanishing Gradient

<https://towardsdatascience.com/intuit-and-implement-batch-normalization-c05480333c5b>

AWS SageMaker Labs

<https://github.com/awslabs/amazon-sagemaker-examples>