Delta Augment - Image Augmentation in the Cloud

Iwan Pettifer-Cole (22476)
University of Bristol
Bristol, UK
iwan.cole.2015@bristol.ac.uk

Reuben Hughes (27401)
University of Bristol
Bristol, UK
rh15406@my.bristol.ac.uk

William Glasse (25449)
University of Bristol
Bristol, UK
wg15313@my.bristol.ac.uk

***Abstract*—As the field of machine learning grows at a rapid pace, computer vision and image recognition form a core use-case of this technology. Data collection is a time consuming process, and remains a bottleneck for Machine Learning projects. Our solution is Delta Augment, a cloud-based service that performs augmentation on image files. Using Amazon Web Services, we provide a highly-available and highly-scalable solution by utilising existing cloud-based solutions.**

Live Demo: rbhu.github.io/DeltaAugment

I. Introduction

Deep learning through using artificial neural networks has been shown to perform exceedingly well on computer vision tasks. Image classification, or more broadly pattern recognition, is a task usually undertaken using supervised learning. This requires pairs of input data and desired output classes, and aims to train an inferred mapping function that correctly classifies unseen input data based on training examples.

One simple approach to improving the performance of a neural network is by increasing its size. This includes increasing the number of nodes per layer, and the number of layers themselves. The main drawback from doing this is the increase in training hyper-parameters, leading to increases in the computational cost *and* is prone to over-fitting. Another approach to improving network performance is exposing it to a larger set of training data. Crawling the internet can be time consuming, and purchasing annotated datasets costly. We can artificially increase existing datasets by performing simple transformations on images, including rotation, cropping, and addition of noise. This is known as data augmentation, and is a technique that when used in the correct proportions [16] can have dramatic increases on network performance.

Data Augmentation can be performed in an 'offline' method, that is *before* training occurs. It can be a computationally expensive process, and we present an implementation of a service that augments image files in the cloud. Cloud computing has been described as the next epoch of transformation in technology [15], and providers such as AWS have accelerated innovation and development by driving down the cost of computation as a utility. Using AWS to abstract away from unnecessary heavy-lifting, we will now discuss our implementation, its current drawbacks, and potential future improvements.

II. Functionality

Delta Augment currently implements a set of features inline with our minimal viable product. The core functionality of our service is a web form that allows image upload with additional metadata, distributed cloud processing of the uploaded files, and returns the output images to the user. Our website is a single page containing information about our service, and a form to upload image files to be augmented. Some input validation is done on the form for UX purposes, however rigorous validation is performed server-side. A grid of all previously uploaded images, and the option to download their corresponding augmentations, is also present. The input form allows users to define a unique string for their upload, some tags related to their image (for future search functionality), and the number of augmentations to perform. At present we limit this to a maximum of 20 augmentations per upload, to maintain a low compute time when using AWS Lambda, and a small output zip file. Currently, the augmentations performed are randomly selected as rotations, flips, or zooms (or a combination of each). Examples of augmented images can be downloaded via our live demo. Once a file is processed a notification is displayed to the user, and if an error occurs, a notification explaining why also appears. When processing is successful, a large download button appears which links to the newly created augmented images. Currently, all images uploaded are publicly accessible. We require users to download a zip file to view the resultant augmentations.

III. Architecture

We chose Amazon Web Services as the platform on which to build our application. Key reasons for this are the ease of inter-operability between AWS components, well documented APIs and extensively granular configuration options. Amazon also provides a reliable global service [14] with rapid failure recovery, and global low latency [6] as a result of strategic placement of data centres in over 20 regions. We also used AWS to avoid re-inventing the wheel: if developing a component ourselves didn't differentiate our application from competition or add value, we used an existing AWS solution to create that component. The level of abstraction provided allowed us to focus our time developing our unique service, instead of configuring VMs and web servers.

Our system architecture is shown in Figure 1. When a user sends a HTTP `GET` request for our site, static stylesheets

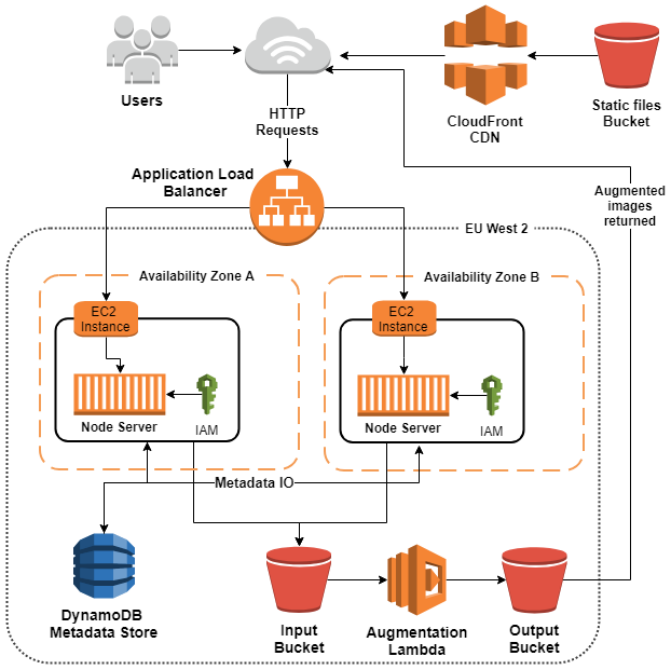

Fig. 1. AWS service architecture diagram

and javascript files are loaded from the nearest "edge location" on Amazon's Content Delivery Network (CDN). Our service exposes two API endpoints, `/getimagelist` and `/upload`. When the first is called, the request is routed through a load balancer to one of two Node servers. It returns a set of images to be displayed on the website homepage. The second API endpoint is called when a user submits an image to be augmented. After again being routed through to one of the Node servers, we validate the request against metadata stored in a DynamoDB table. This is used to store unique IDs and corresponding image tags. The original image is then stored in a source S3 bucket `img-bucket-irw`, triggering a call to a Lambda function. The Lambda processes the file and augments it, returning the zipped images to a sink S3 bucket `img-bucket-irw-augmented`. The user is presented with a download link in the form of a popup button, and the process is complete. Subsequent `GET` requests to the homepage also retrieve thumbnails of previously augmented images from the source bucket, and display them in a grid. We will now discuss the design considerations we made for each component, and how we focused on their ability to scale.

A. *NodeJS*

NodeJS is a single-threaded, event-loop web server, which has demonstrated its ability to handle many thousand [12], [17] concurrent requests simultaneously. We manage the underlying OS that the Node server runs on, and therefore it is part of an IaaS stack.

The initial index `GET` request to Node responds near instantaneously, as discussed in a later section. A `POST` request to the `/upload` endpoint however takes more time, since it handles I/O to S3 and DynamoDB. Using a simple node script for basic load testing, we profiled the runtime of this endpoint. This profiling does not measure our lambda, only uploading. Over 100 trials we found the average response time for successful requests to be 47ms. We noticed spikes in this runtime occurred when the endpoint had been idle for a period of time. Subsequent requests however returned much quicker, suggesting some warm-up time on initially calling S3 and DynamoDB APIs. If Node *did* block, we could extrapolate this figure to approximately 72,000 sequential uploads per hour ($\sim$1 request per 50ms). Since Node doesn't block, we believe this figure might be higher in reality. A single Node server alone isn't our approach to scalability, but as a core aspect of our application it is an excellent foundation to build upon.

B. *Elastic Compute Cloud*

Elastic Compute Cloud (EC2) [4] is one of Amazon's IaaS services allowing users to run fully customisable VMs, ranging from single core low memory to hundreds of cores, multi-terabyte memory. EC2 is a form of utility computing, where you pay-as-you-go for compute power. An EC2 "instance" runs an Amazon Machine Image (AMI), which can come with a pre-configured OS and pre-installed applications. We opted for an Ubuntu image with a web stack installed. Selecting an appropriate region to deploy instances within is an important consideration. We foresee that the majority of our users will be within the United Kingdom, and so selected `EU West 2` as our region. Aside from the OS, we only require a small amount of backing storage as our service's data is hosted in S3. Attached to our EC2 instances are "security groups", which define firewall rules allowing SSH connections and incoming and outgoing HTTP traffic. This level of granular control is what makes EC2 an appealing service to use compared to other PaaS offerings. AMI's can be cloned from snapshots of running instances, and this makes spawning new identical instances trivial.

C. *Simple Storage Service*

Simple Storage Service [11] (S3) is Amazon's cloud based object storage service, which operates to a baseline standard of four 9's availability. It can store objects up to 5TB in size, has extensive metadata tagging configurations, a number of access control options. Objects can be searched using prefixes, and are stored in "buckets" which have a flat hierarchical structure. As with many other AWS components, S3 operates on a pay-as-you-use model. Due to Amazon's scale, it advertises theoretically no maximum storage limit. While S3 has multiple backup and distribution options for added redundancy, we opt not to use these as users will primarily download their outputs only once (hence loss of data is not a major concern).

We use three buckets as part of our service. The first is an input bucket for user uploaded images, called `img-bucket-irw`. This bucket has access control policies that allow public read-only access to images stored, for the purpose of displaying the image grid on our webpage. Another

policy prevents public writing, but allows our EC2 instances to perform `PUT` requests to store user uploaded images. Our EC2 instances contain secret IAM keys, which enable this behaviour. The second bucket contains the results of the augmentation, called `img-bucket-irw-augmented`. This bucket again has public read-only access, which enables downloading of ZIP files through the web interface. Both of these buckets have "lifecycle rules" applied to all objects within them. Our rule expires an object 27 days after its creation, deleting it. Since source images and output ZIPs are created approximately simultaneously, this lifecycle rule will ensure deletions are essentially synchronised between the two buckets. This reduces our cost of storage, and improves page load speed by periodically clearing old uploads. As our service scales, this is a simple way of maintaining a smaller footprint.

The third bucket is used to host static files for the webpage itself, and is called `irw-files`. We include static images, stylesheets and scripts in this bucket. When Angular builds our front-end, we modify the `<src>` and `<href>` tags within `index.html` to point to this S3 bucket. This reduces the bandwidth used by our EC2 instance drastically. In the next section, we discuss how we made further networking performance boosts by distributing this content on a CDN.

D. *CloudFront*

CloudFront [1] is Amazon's proprietary Content Distribution Network (CDN) with over 149 *Points of Presence* globally. This global low-latency network aims to deliver both static and dynamic content to end-users at a greater speed than with regular distribution. As discussed previously, many components of our client-facing service are static, and do not change frequently. In addition to using our S3 Bucket `irw-files` to host this static content, we use CloudFront to distribute the content of this bucket across the globe. When a user makes a request, static content that passes through an edge location is cached, reducing latency of subsequent requests. This also has the added bonus of reducing load on the origin server, our EC2 instances. To highlight this, an initial request to load Delta Augment (with no pre-loaded images) is 896KB. Of this, only the initial request for the `index.html` reaches the origin, and is itself 1.4KB. Aside from requests made to Bootstrap and jQuery resources (that themselves are served from other CDNs), 847KB's of other requested data is forwarded through CloudFront to S3. This means that 99.8% of the request traffic avoids our origin server. This reduces the bandwidth I/O on our EC2 instances, allowing more bandwidth for users to upload their datasets. This also releases wasted compute time serving static files, enabling our Node server to handle more requests. While individually a small saving, over the course of 10,000 requests this results in 8.47GB of traffic avoiding our EC2 instance. CloudFront is an essential component in order for our service to scale globally.

E. *Elastic Load Balancing*

Amazon's Elastic Load Balancing [9] (ELB) service is able to automatically distribute incoming traffic across a number of endpoints, in our case EC2 instances. A key ELB feature is health checks. Detecting when a target is not performing optimally, it can divert traffic until a target regains health. We use the Application Load Balancer, and define our target group as two identical EC2 instances running within one region, but on *two* separate Availability Zones. Once activated we modified the security groups on our EC2 instances to block traffic external to AWS. This means only internal traffic (ELB to EC2) is accepted by our Node servers. This forces all public traffic through ELB, and we used CloudWatch to monitor the split of traffic between AZs to ensure correct load balancing.

F. *Lambda*

Lambda [7] is a serverless compute platform which executes code in response to an event being triggered. It is considered a "Function as a Service", and is ideal for running on-demand functions that are called in irregular intervals. Abstracting away from running a server, Lambdas are highly efficient at handling sudden spikes in invocation. They have a defined reserved memory capacity, and a theoretically "unlimited" number of concurrent invocations. This means Lambda implicitly scales. When called initially, Lambda retains all the code needed for execution in memory for an undisclosed cool-down period. During this time, subsequent calls will execute much faster compared to loading everything from backing store.

Augmenting images is *the* core component of our service, requiring us to ensure that this step isn't a bottleneck. We augment using a single python script which takes an input file, processes it, and outputs the results. It has no dependency on other images being processed, making it ideal for Lambda as it is a highly parallelisable task. We used the open-source `Augmentor` python library [13]. Lambda's are triggered by events, and we define our trigger to be a new image uploaded into the `img-bucket-irw` bucket. We use metadata attached to the image to determine the number of augmentations to perform. Using temporary memory allocated, we compress the outputs and upload them to our output bucket. See *Section IV* for our research into Lambda performance.

G. *DynamoDB*

DynamoDB [2] is Amazon's proprietary non-relational database platform, and is a NoSQL database. A main feature of this service is that it charges based on data throughput rather than storage-capacity, ideal for situations where the volume of queries can vary greatly. It performs well at scale, with microsecond latency due to an in-memory cache system. We use a DynamoDB table to store metadata about user uploads, including a UID, URL of images, and tags. Rigorous validation occurs before we store the data. When uploading an image, we check that a user submitted UID does not collide with an existing UID, which prevents S3 data overwrites. While

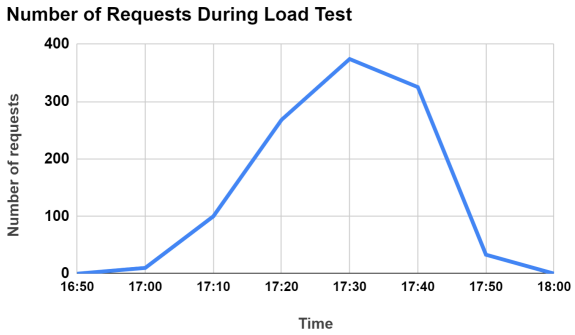

Fig. 2. Number of requests during a load test on our Lambda function

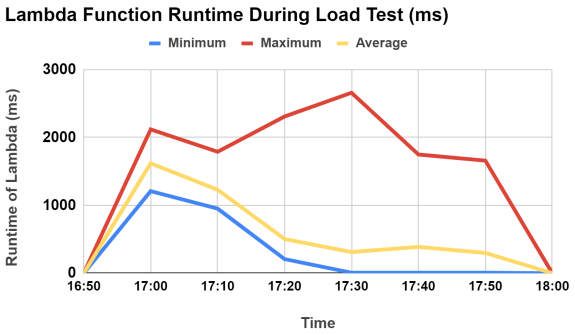

Fig. 3. Runtime of our Lambda function during load test

not currently implemented, we store 'tags' on images to allow future development of a search feature in our service.

IV. Scalability

Tying together all our scalability features, we first focus on Node. As it's single threaded, memory and CPU power can become a bottleneck quickly. To initially raise the capacity of a single Node server, we can allocate it more memory and a larger processor on its instance. On an EC2 instance, we can then enable "Auto Scaling" with a minimum and maximum number of clone instances. This dynamically provisions instances according to demand. We set the max to four, and use a network policy to keep them within *one* Availability Zone (AZ). This means that there is capacity provisioned for increases in traffic. Next, we clone the parent instance, and create a copy in a *separate* AZ. This contains the same auto-scaling policy, but now ensures uptime in the event of a whole AZ failing. We pair the two instances using a "target group", and point our ELB at them to distribute traffic. This is essential for creating High Availability. This approach is known as "horizontal scaling", and our target group used by ELB can contain up to 1000 [5] instances. A future improvement would be to allow ELB to dynamically create more instances as required. As discussed previously, CloudFront drastically reduces load on the instances themselves, forming a core part of our ability to scale.

Our Lambda function is defined to have no upper limit on capacity. We load tested it with an increasing number of input images over the space of an hour (Figure 2), and used CloudWatch metrics to analyse the corresponding runtimes (Figure 3). From a cold-start, Lambda loads all the libraries from backing store and keeps them cached for less time [8]. This explains the initial spike in function runtime. Our load test peaks at 17:30, with just under 400 lambda invocations. At this point, the average runtime of a lambda call reaches its lowest value of 7.52ms. The runtime remains fairly constant after this, indicating that Lambda is heavily caching our function, and the cool-off period before unloading has been extended. This demonstrates that Lambda will still perform under a heavy load during spikes, or general increases in traffic.

V. Further Work

First, we will focus on improvements related to scalability. Our first focus will be switching from EC2 to Elastic Beanstalk (EB) [3]. EB is a layer of abstraction above EC2, and runs application code in place of VMs. This makes it a Platform as a Service. It allows more flexible scaling options, and removes the complexity of configuring our VMs, such as maintaining correct versions of npm and Node.

Another improvement would be cross-region scaling. To implement this, we would use Route53 [10], Amazon's highly scalable DNS service. Route 53 can resolve domain names dynamically to targets with the lowest latency. After deploying EC2 instances in many regions globally, it would intelligently route requests to a load balancer in the nearest corresponding region. In Figure 1, the box labelled "EU West 2" would be duplicated across regions, and Route 53 would connect the internet component to multiple Application Load Balancers. This would help achieve global availability of our service.

In terms of the application itself, we would like to add more user options in regards to the augmentations taking place. This would mean more "upload form" components, such as options to disable image-flipping, or a slider for addition of noise. We would attach this additional information as metadata when uploading an image to the input S3 bucket, which the Lambda would then parse and augment as appropriate.

Another user-focused feature would be search functionality, enabled through our existing DynamoDB table. User inputted search queries would target a new API endpoint (e.g. `/search?q=cat`), with the Node server extracting the URL parameter 'q'. The server would then query the DynamoDB table for entries with "tags" matching the query, returning the original URLs as appropriate.

VI. Conclusion

In this report we have presented Delta Augment, an online image augmentation tool running on AWS. We have demonstrated our architecture's ability to scale, provided rationale behind our design considerations, and outlined future improvements for further scalability.

REFERENCES

[1] Aws cloudfront. www.aws.amazon.com/cloudfront/features.

[2] Aws dynamodb. www.aws.amazon.com/dynamodb/features.

[3] Aws elastic beanstalk. www.aws.amazon.com/elasticbeanstalk/details.

[4] Aws elastic compute cloud. www.aws.amazon.com/EC2/features.

[5] Aws elb limts. docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-limits.html.

[6] Aws global infrastructure. www.aws.amazon.com/about-aws/global-infrastructure.

[7] Aws lambda. www.aws.amazon.com/lambda/features.

[8] Aws lambda function cache time. https://read.acloud.guru/how-long-does-aws-lambda-keep-your-idle-functions-around-before-a-cold-start-bf715d3b810.

[9] Aws load balancing. www.aws.amazon.com/elasticloadbalancing.

[10] Aws route 53. www.aws.amazon.com/route53.

[11] Aws simple storage service. www.aws.amazon.com/s3/features.

[12] Nodejs with 205k concurrent connections. http://blog.caustik.com/2012/04/10/node-js-w250k-concurrent-connections/. Accessed: 2018-12-18.

[13] Python augmentor. https://github.com/mdbloice/Augmentor.

[14] Amazon, Inc. *AWS Reliability Pillar Whitepaper*, September 2018.

[15] Nicholas Carr. *The Big Switch: Rewiring the World, from Edison to Google*, page 12. W. W. Norton Co., 2009.

[16] L Perez and J Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*.

[17] D Synodinos and R Dahl. Deep inside nodejs. https://www.infoq.com/interviews/node-ryan-dahl, Dec 2010. InfoQ: QCon Conference.