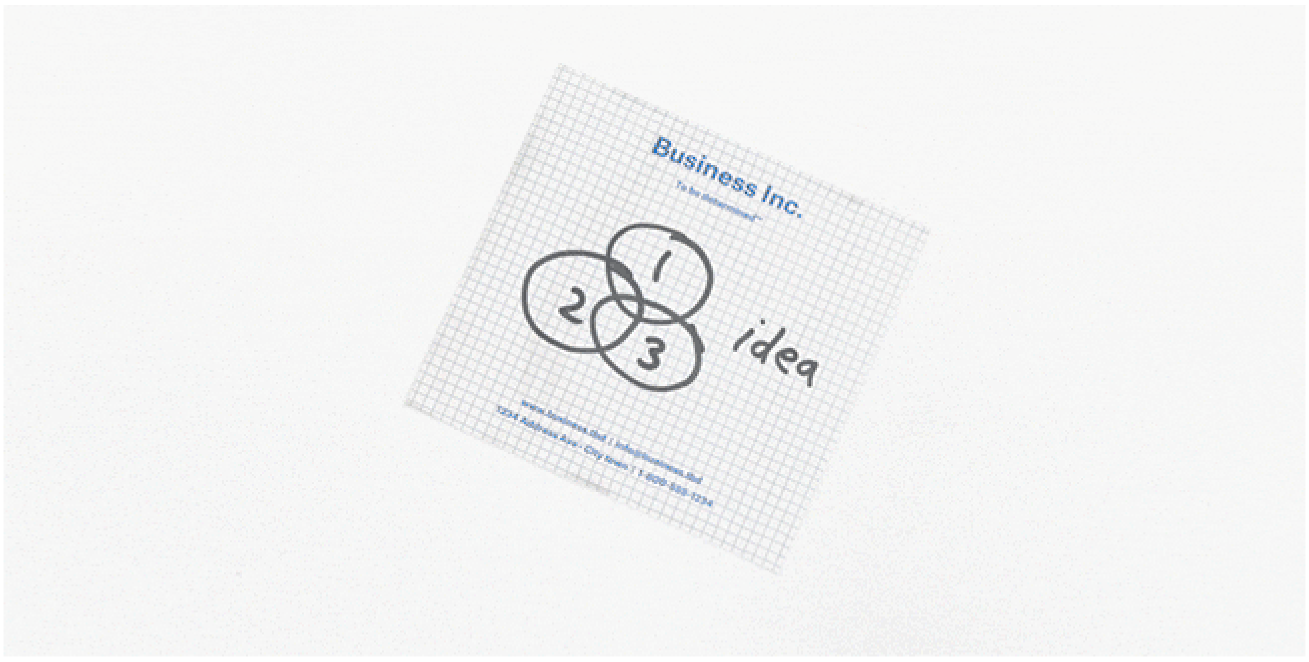
Creating a Modern OCR Pipeline Using Computer Vision and Deep Learning

Brad Neuberg | April 12, 2017

In this post we will take you behind the scenes on how we built a state-of-the-art Optical Character Recognition (OCR) pipeline for our mobile document scanner. We used computer vision and deep learning advances such as bi-directional Long Short Term Memory (LSTMs), Connectionist Temporal Classification (CTC), convolutional neural nets (CNNs), and more. In addition, we will also dive deep into what it took to actually make our OCR pipeline production-ready at Dropbox scale.



In previous posts we have described how Dropbox’ s mobile document scanner works. The document scanner makes it possible to use your mobile phone to take photos and “scan” items like receipts and invoices. Our mobile document scanner only outputs an image — any text in the image is just a set of pixels as far as the computer is concerned, and can’t be copy-pasted, searched for, or any of the other things you can do with text.

Hence the need to apply Optical Character Recognition, or OCR. This process extracts actual text from our doc-scanned image. Once OCR is run, we can then enable the following features for our Dropbox Business users:

- Extract all the text in scanned documents and index it, so that it can be searched for later
- Create a hidden overlay so text can be copied and pasted from the scans saved as PDFs

When we built the first version of the mobile document scanner, we used a commercial off-the-shelf OCR library, in order to do product validation before diving too deep into creating our own machine learning-based OCR system. This meant integrating the commercial system into our scanning pipeline, offering both features above to our business users to see if they found sufficient use from the OCR. Once we confirmed that there was indeed strong user demand for the mobile document scanner and OCR, we decided to build our own in-house OCR system for several reasons.

First, there was a cost consideration: having our own OCR system would save us significant money as the licensed commercial OCR SDK charged us based on the number of scans. Second, the commercial system was tuned for the traditional OCR world of images from flat bed scanners, whereas our operating scenario was much tougher, because mobile phone photos are far more unconstrained, with crinkled or curved documents, shadows and uneven lighting, blurriness and reflective highlights, etc. Thus, there might be an opportunity for us to improve recognition accuracy.

In fact, a sea change has happened in the world of computer vision that gave us a unique opportunity. Traditionally, OCR systems were heavily pipelined, with hand-built and highly-tuned modules taking advantage of all kinds of conditions they could assume to be true for images captured using a flatbed scanner. For example, one module might find lines of text, then the next module would find words and segment letters, then another module might apply different techniques to each piece of a character to figure out what the character is, etc. Most methods rely on binarization of the input image as an early stage, and this can be brittle and discards important cues. The process to build these OCR systems was very specialized and labor intensive, and the systems could generally only work with fairly constrained imagery from flat bed scanners.

The last few years has seen the successful application of deep learning to numerous problems in computer vision that have given us powerful new tools for tackling OCR without having to replicate the complex processing pipelines of the past, relying instead on large quantities of data to have the system automatically learn how to do many of the previously manually-designed steps.

Perhaps the most important reason for building our own system is that it would give us more control over own destiny, and allow us to work on more innovative features in the future.

In the rest of this blog post we will take you behind the scenes of how we built this pipeline at Dropbox scale. Most commercial machine learning projects follow three major steps:

1. Research and prototyping to see if something is possible
2. Productionization of the model for actual end users
3. Refinement of the system in the real world

We will take you through each of these steps in turn.

Research and Prototyping

Our initial task was to see if we could even build a state of the art OCR system at all.

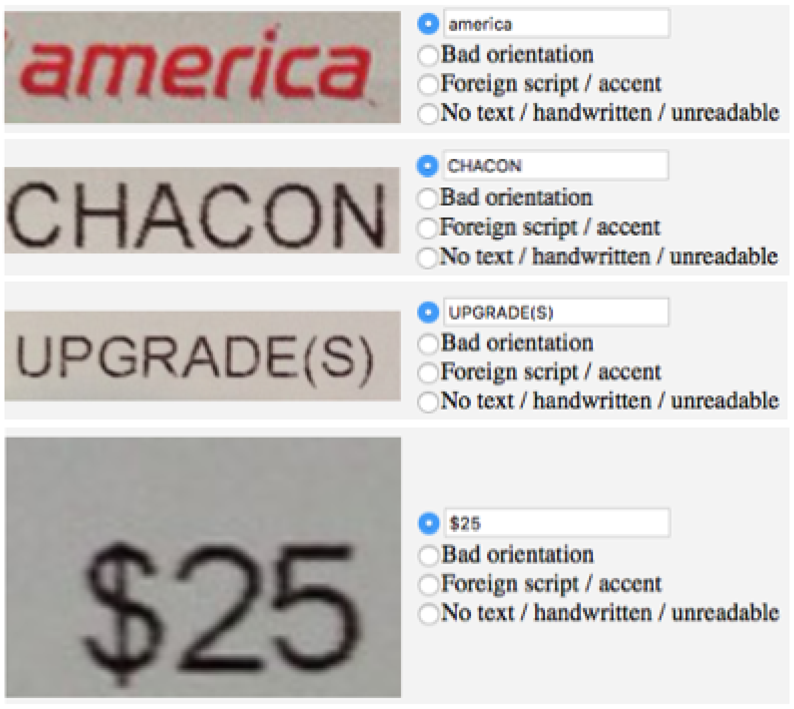
We began by collecting a representative set of donated document images that match what users might upload, such as receipts, invoices, letters, etc. To gather this set, we asked a small percentage of users whether they would donate some of their image files for us to improve our algorithms. At Dropbox, we take user privacy very seriously and thus made it clear that this was completely optional, and if donated, the files would be kept private and secure. We use a wide variety of safety precautions with such user-donated data, including never keeping donated data on local machines in permanent storage, maintaining extensive auditing, requiring strong authentication to access any of it, and more.

Another important, machine learning-specific component for user-donated data is how to label it. Most current machine learning techniques are strongly-supervised, meaning that they require explicit manual labeling of input data so that the algorithms can learn to make predictions themselves. Traditionally, this labeling is done by outside workers, often using a micro-work platform such as Amazon’s Mechanical Turk (MTurk). However, a downside to using MTurk is that each item might be seen and labeled by a different worker, and we certainly don’t want to expose user-donated data in the wild like this!

Thus, our team at Dropbox created our own platform for data annotation, named DropTurk. DropTurk can submit labeling jobs either to MTurk (if we are dealing with public non-user data) or a small pool of hired contractors for user-donated data. These contractors are under a strict non-disclosure agreement (NDA) to ensure that they cannot keep or share any of the data they label. DropTurk contains a standard list of annotation task UI templates that we can rapidly assemble and customize for new datasets and labeling tasks, which enables us to annotate our datasets quite fast.

For example, here is a DropTurk UI meant to provide ground truth data for individual word images, including one of the following options for the workers to complete:

- Transcribing the actual text in an image
- Marking whether the word is oriented incorrectly
- Marking whether it’s a non-English script
- Marking whether it’s unreadable or contains no text



[✓ america] ○ Bad orientation ○ Foreign script / accent ○ No text / handwritten / unreadable

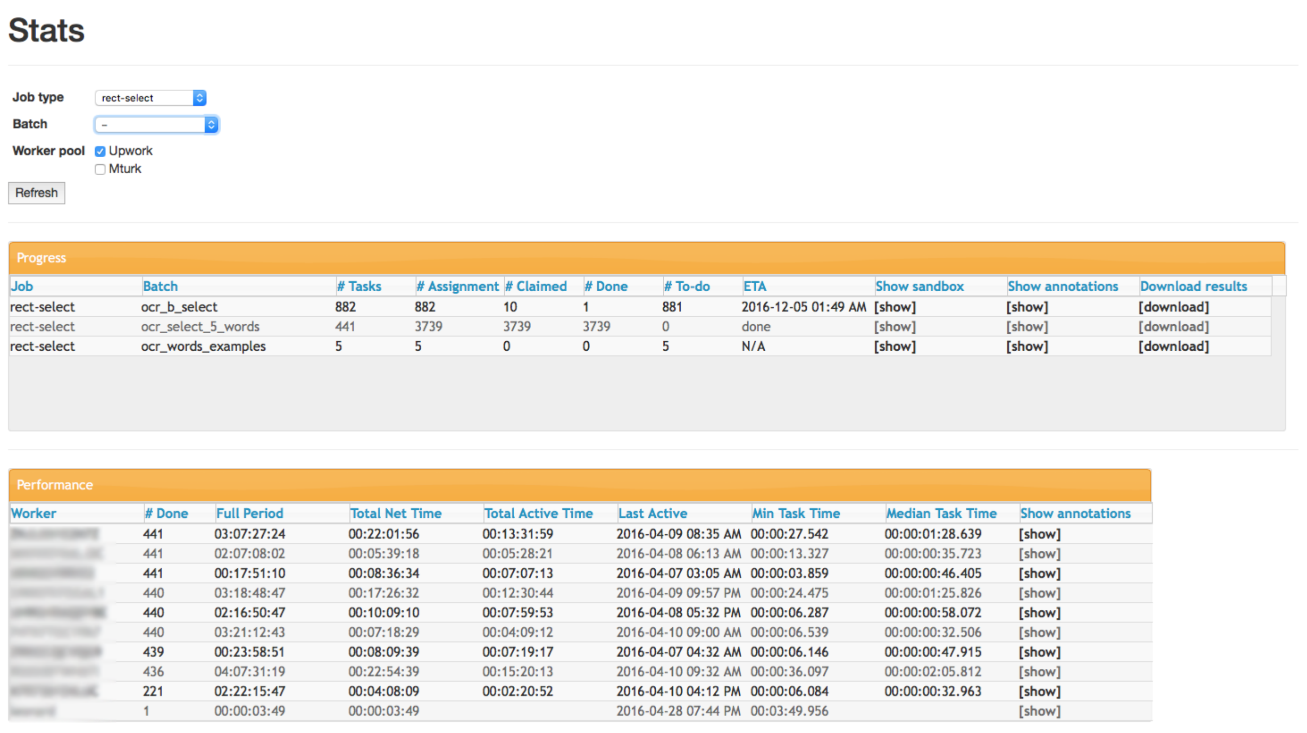
[✓ CHACON] ○ Bad orientation ○ Foreign script / accent ○ No text / handwritten / unreadable

[✓ UPGRADE(S)] ○ Bad orientation ○ Foreign script / accent ○ No text / handwritten / unreadable

[✓ $25] ○ Bad orientation ○ Foreign script / accent ○ No text / handwritten / unreadable

DropTurk UI for adding ground truth data for word images

Our DropTurk platform includes dashboards to get an overview of past jobs, watch the progress of current jobs, and access the results securely. In addition, we can get analytics to assess workers' performance, even getting worker-level graphical monitoring of annotations of ongoing jobs to catch potential issues early on:



Stats

Job type: rect-select

Batch: –

Worker pool: [✓ Upwork] Mturk

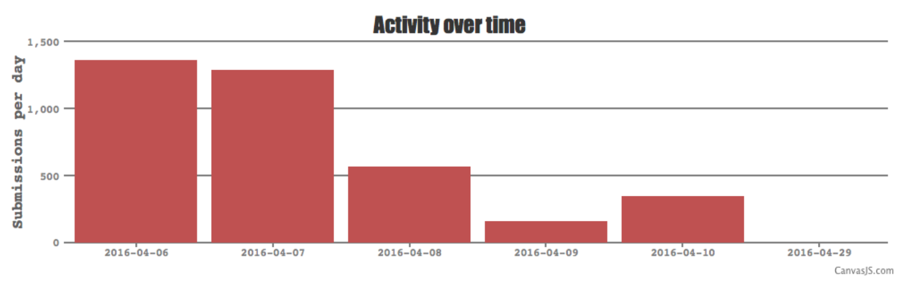
Refresh

Progress

Job	Batch	# Tasks	# Assignment	# Claimed	# Done	# To-do	ETA	Show sandbox	Show annotations	Download results
rect-select	ocr_b_select	882	882	10	1	881	2016-12-05 01:49 AM	[show]	[show]	[download]
rect-select	ocr_select_5_words	441	3739	3739	3739	0	done	[show]	[show]	[download]
rect-select	ocr_words_examples	5	5	0	0	5	N/A	[show]	[show]	[download]

Performance

Worker	# Done	Full Period	Total Net Time	Total Active Time	Last Active	Min Task Time	Median Task Time	Show annotations
[redacted]	441	03:07:27:24	00:22:01:56	00:13:31:59	2016-04-09 08:35 AM	00:00:27.542	00:00:01:28.639	[show]
[redacted]	441	02:07:08:02	00:05:39:18	00:05:28:21	2016-04-08 06:13 AM	00:00:13.327	00:00:00:35.723	[show]
[redacted]	441	00:17:51:10	00:08:36:34	00:07:07:13	2016-04-07 03:05 AM	00:00:03.859	00:00:00:46.405	[show]
[redacted]	440	03:18:48:47	00:17:26:32	00:12:30:44	2016-04-09 09:57 PM	00:00:24.475	00:00:01:25.826	[show]
[redacted]	440	02:16:50:47	00:10:09:10	00:07:59:53	2016-04-08 05:32 PM	00:00:06.287	00:00:00:58.072	[show]
[redacted]	440	03:21:12:43	00:07:18:29	00:04:09:12	2016-04-10 09:00 AM	00:00:06.539	00:00:00:32.506	[show]
[redacted]	439	00:23:58:51	00:08:09:39	00:07:19:17	2016-04-07 04:32 AM	00:00:06.146	00:00:00:47.915	[show]
[redacted]	436	04:07:31:19	00:22:54:39	00:15:20:13	2016-04-10 09:32 AM	00:00:36.097	00:00:02:05.812	[show]
[redacted]	221	02:22:15:47	00:04:08:09	00:02:20:52	2016-04-10 04:12 PM	00:00:06.084	00:00:00:32.963	[show]
[redacted]	1	00:00:03:49	00:00:03:49		2016-04-28 07:44 PM	00:03:49.956		[show]



DropTurk Dashboard

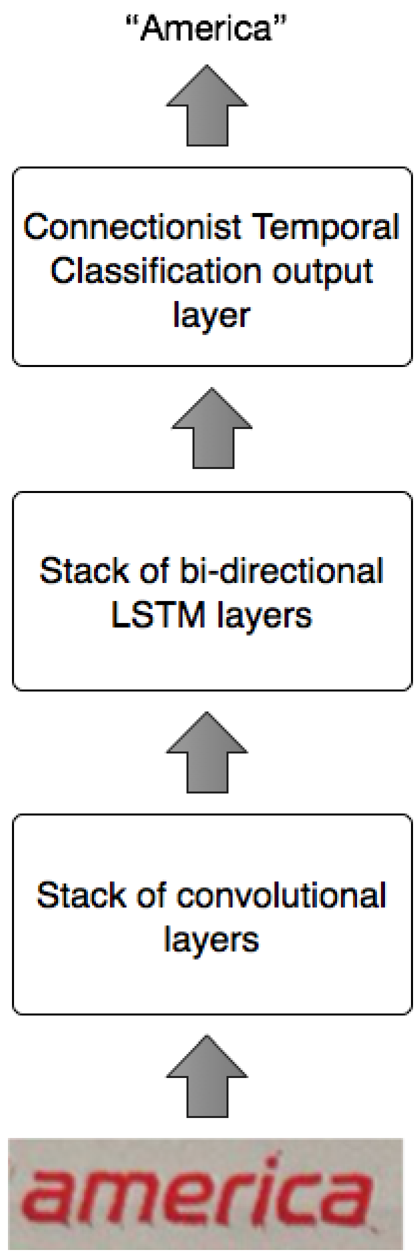
Using DropTurk, we collected both a word-level dataset, which has images of individual words and their annotated text, as well as a full document-level dataset, which has images of full documents (like receipts) and fully transcribed text. We used the latter to measure the accuracy of existing state-of-the-art OCR systems; this would then inform our efforts by telling us the score we would have to meet or beat for our own system. On this particular dataset, the accuracy percentage we had to achieve was in the mid-90s.

Our first task was to determine if the OCR problem was even going to be solvable in a reasonable amount of time. So we broke the OCR problem into two pieces. First, we would use computer vision to take an image of a document and segment it into lines and words; we call that the Word Detector. Then, we would take each word and feed it into a deep net to turn the word image into actual text; we call that the Word Deep Net.

We felt that the Word Detector would be relatively straightforward, and so focused our efforts first on the Word Deep Net, which we were less sure about.

Word Deep Net

The Word Deep Net combines neural network architectures used in computer vision and automatic speech recognition systems. Images of cropped words are fed into a Convolutional Neural Net (CNN) with several convolutional layers. The visual features that are output by the CNN are then fed as a sequence to a Bidirectional LSTM (Long Short Term Memory) — common in speech recognition systems — which make sense of our word "pieces," and finally arrives at a text prediction using a Connectionist Temporal Classification (CTC) layer. Batch Normalization is used where appropriate.


"America"
Connectionist Temporal Classification output layer
Stack of bi-directional LSTM layers
Stack of convolutional layers
america

OCR Word Deep Net

Once we had decided on this network architecture for turning an image of a single word into text, we then needed to figure out how to collect enough data to train it. Deep learning systems typically need huge amounts of training data to achieve good recognition performance; in fact, the amount of training data is often the most significant bottleneck in current systems. Normally, all this data has to be collected and then labeled manually, a time-consuming and expensive process.

An alternative is to programmatically generate training data. However, in most computer vision problems it’ s currently too difficult to generate realistic-enough images for training algorithms: the variety of imaging environments and transformations is too varied to effectively simulate. (One promising area of current research is Generative Adversarial Networks (GANs), which seem to be well-suited to generating realistic data.) Fortunately, our problem in this case is a perfect match for using synthetic data, since the types of images we need to generate are quite constrained and can thus be rendered automatically. Unlike images of natural or most manmade objects, documents and their text are synthetic and the variability of individual characters is relatively limited.

Our synthetic data pipeline consists of three pieces:

1. A corpus with words to use
2. A collection of fonts for drawing the words
3. A set of geometric and photometric transformations meant to simulate real world distortions

The generation algorithm simply samples from each of these to create a unique training example.

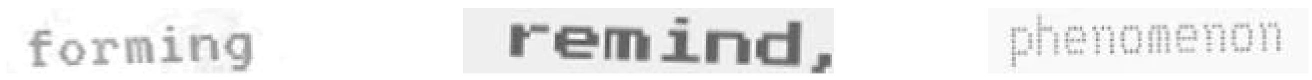
Synthetically generated word images

We started simply with all three, with words coming from a collection of Project Gutenberg books from the 19th century, about a thousand fonts we collected, and some simple distortions like rotations, underlines, and blurs. We generated about a million synthetic words, trained our deep net, and then tested our accuracy, which was around 79%. That was okay, but not good enough.

Through many iterations, we evolved each piece of our synthetic data pipeline in many ways to improve the recognition accuracy. Some highlights:

- We noticed that we weren’t doing well on receipts, so we expanded our word corpus to include the Uniform Product Code (UPC) database, which has entries like “24QT TISSUE PAPER” which commonly occur on receipts.
- We noticed the network was struggling with letters with disconnected segments. This revealed a deeper problem: receipts are often printed with thermal fonts that have stippled, disconnected, or ink smudged letters, but our network had only been given training data with smooth continuous fonts (like

from a laser printer) or lightly bit-mapped characters (like in a screenshot). To address this shortcoming, we eventually tracked down a font vendor in China who could provide us with representative ancient thermal printer fonts.

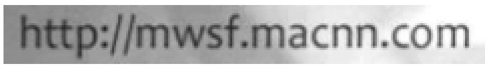


Synthetically generated words using different thermal printer fonts, common in receipts

- Our font selection procedure was too naive initially. We ended up hand-selecting about 2,000 fonts. Not all fonts are used equally. So, we did research on the top 50 fonts in the world and created a font frequency system that allowed us to sample from common fonts (such as Helvetica or Times New Roman) more frequently, while still retaining a long tail of rare fonts (such as some ornate logo fonts). In addition, we discovered that some fonts have incorrect symbols or limited support, resulting in just squares, or their lower or upper case letters are mismatched and thus incorrect. We had to go through all two thousand fonts by hand and mark those that had invalid symbols, numbers, or casing, so that we didn’ t inadvertently train the network with incorrect data.
- The upstream Word Detector (described later) was tuned to provide high recall and low precision. It was overzealous in finding text in images so that it wouldn’ t miss any of the actual text (high recall) at the expense of often “finding” words that weren’ t actually there (low precision). This meant that the Word Deep Net had to deal with a very large number of essentially empty images with noise. So we had our synthetic data pipeline generate representative negative training examples with empty ground truth strings, including common textured backgrounds, like wood, marble countertops, etc.

Synthetically generated negative training examples

- From a histogram of the synthetically generated words, we discovered that many symbols were underrepresented, such as / or &. We artificially boosted the frequency of these in our synthetic corpus, by synthetically generating representative dates, prices, URLs, etc.
- We added a large number of visual transformations, such as warping, fake shadows, and fake creases, and much more.

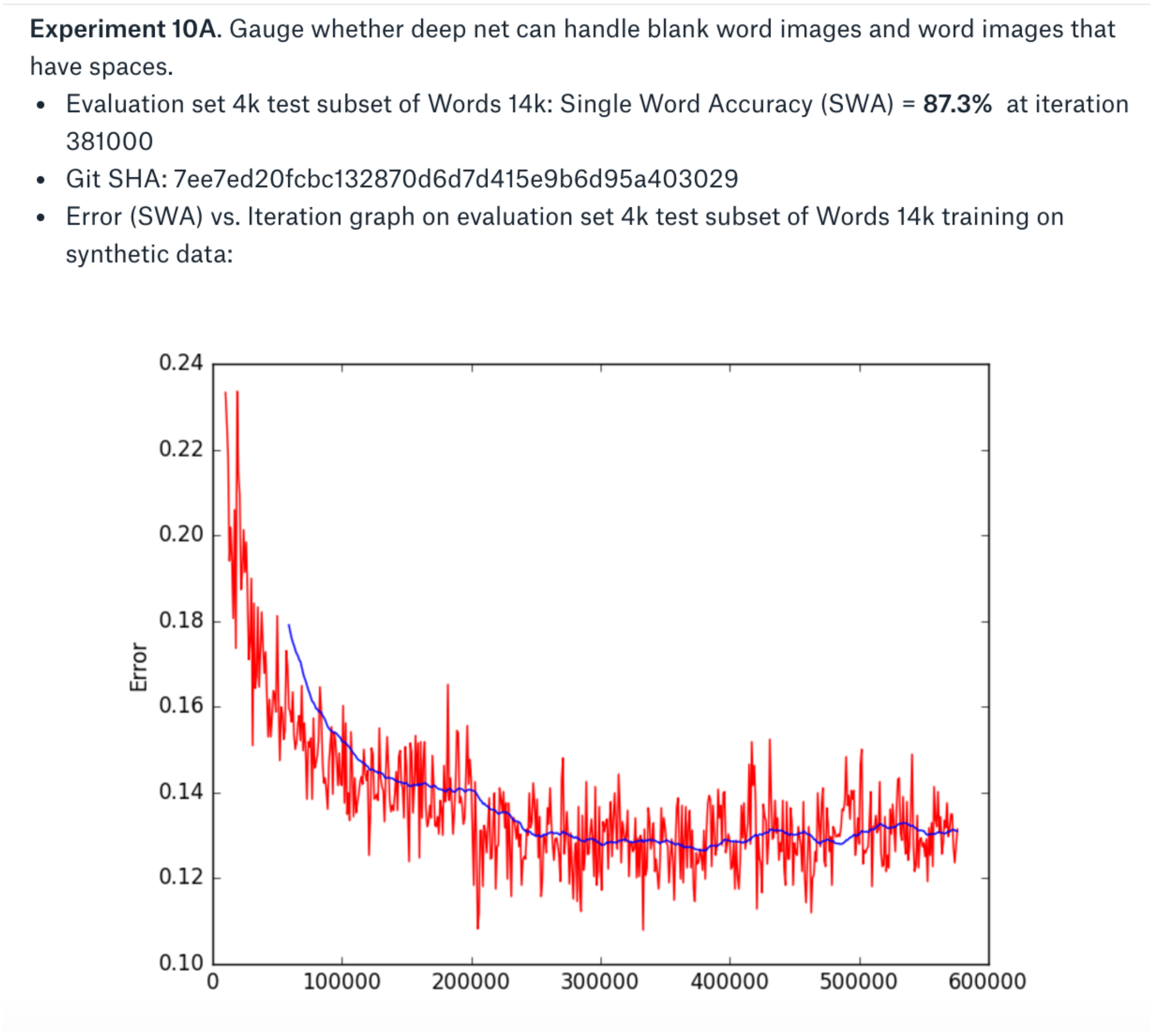


Fake shadow effect

Data is as important as the machine learning model used, so we spent a great deal of time refining this data generation pipeline. At some point, we will open source and release this synthetically generated data for others to train and validate their own systems and research on.

We trained our network on Amazon EC2 G2 GPU instances, spinning up many experiments in parallel. All of our experiments went into a lab notebook that included everything necessary to replicate experiments so we could track unexpected accuracy bumps or losses.

Our lab notebook contained numbered experiments, with the most recent experiment first. It tracked everything needed for machine learning reproducibility, such as a unique git hash for the code that was used, pointers to S3 with generated data sets and results, evaluation results, graphs, a high-level description of the goal of that experiment, and more. As we built our synthetic data pipeline and trained our network, we also built many special purpose tools to visualize fonts, debug network guesses, etc.



Experiment 10A. Gauge whether deep net can handle blank word images and word images that have spaces.

- Evaluation set 4k test subset of Words 14k: Single Word Accuracy (SWA) = **87.3%** at iteration 381000
- Git SHA: 7ee7ed20fcbc132870d6d7d415e9b6d95a403029
- Error (SWA) vs. Iteration graph on evaluation set 4k test subset of Words 14k training on synthetic data:

Example early experiment tracking error rate vs. how long our Word Deep Net had trained, against an evaluation dataset that consisted of just single words (Single Word Accuracy)

Our early experiments tracked how well Word Deep Net did on OCR-ing images of single words, which we called Single Word Accuracy (SWA). Accuracy in this context meant how many of the ground truth words the deep net got right. In addition, we tracked precision and recall for the network. Precision refers to the fraction of words returned by the deep net that were actually correct, while recall refers to the fraction of evaluation data that is correctly predicted by the deep net. There tends to be a tradeoff between precision and recall.

For example, imagine we have a machine learning model that is designed to classify an email as spam or not. Precision would be whether all the things that were labeled as spam by the classifier, how many were actually spam? Recall, in contrast, would be whether of all the things that truly are spam, how many did we label? It is possible to correctly label spam emails (high precision) while not actually labeling all the true spam emails (low recall).

Week over week, we tracked how well we were doing. We divided our dataset into different categories, such as `register_tapes` (receipts), `screenshots`, `scanned_docs`, etc., and computed accuracies both individually for each category and overall across all data. For example, the entry below shows early work in our lab notebook for our first full end-to-end test, with a real Word Detector coupled to our real Word Deep Net. You can see that we did pretty terribly at the start:

Screenshot from early end-to-end experiments in our lab notebook

At a certain point our synthetic data pipeline was resulting in a Single Word Accuracy (SWA) percentage in the high-80s on our OCR benchmark set, and we decided we were done with that portion. We then collected about 20,000 real images of words (compared to our 1 million synthetically generated words) and used these to fine tune the Word Deep Net. This took us to an SWA in the mid-90s.

We now had a system that could do very well on individual word images, but of course a real OCR system operates on images of entire documents. Our next step was to focus on the document-level Word Detector.

Word Detector

For our Word Detector we decided to not use a deep net-based approach. The primary candidates for such approaches were object detection systems, like RCNN, that try to detect the locations (bounding boxes) of objects like dogs, cats, or plants from images. Most images only have perhaps one to five instances of a given object.

However, most documents don’ t just have a handful of words — they have hundreds or even thousands of them, i.e., a few orders of magnitude more objects than most neural network-based object detection systems were capable of finding at the time. We were thus not sure that such algorithms would scale up to the level our OCR system needed.

Another important consideration was that traditional computer vision approaches using feature detectors might be easier to debug, as neural networks as notoriously opaque and have internal representations that are hard to understand and interpret.

We ended up using a classic computer vision approach named Maximally Stable Extremal Regions (MSERs), using OpenCV’ s implementation. The MSER algorithm finds connected regions at different thresholds, or levels, of the image. Essentially, they detect blobs in images, and are thus particularly good for text.

Our Word Detector first detects MSER features in an image, then strings these together into word and line detections. One tricky aspect is that our word deep net accepts fixed size word image inputs. This requires the word detector to thus sometimes include more than one word in a single detection box, or chop a single word in half if it is too long to fit the deep net’ s input size. Information on this chopping then has to be propagated through the entire pipeline, so that we can re-assemble it after the deep net has run. Another bit of trickiness is dealing with images with white text on dark backgrounds, as opposed to dark text on white backgrounds, forcing our MSER detector to be able to handle both scenarios.

Combined End-to-End System

Once we had refined our Word Detector to an acceptable point, we chained it together with our Word Deep Net so that we could benchmark the entire combined system end-to-end against document-level images rather than our older Single Word Accuracy benchmarking suite. However, when we first measured the end-to-end accuracy, we found that we were performing around 44% — quite a bit worse than the competition.

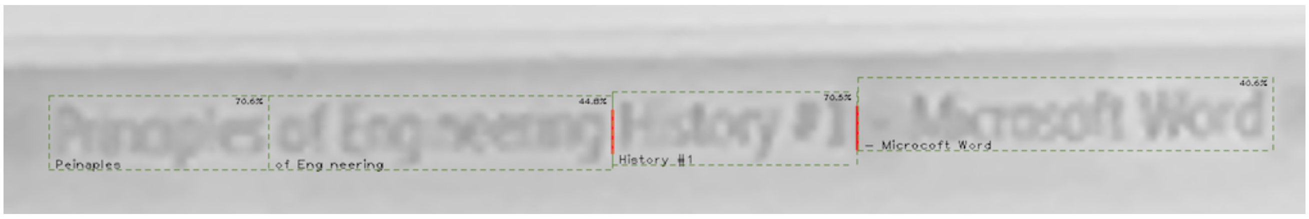
The primary issues were spacing and spurious garbage text from noise in the image. Sometimes we would incorrectly combine two words, such as “*helloworld*”, or incorrectly fragment a single word, such as “*wo rld*”.

Our solution was to modify the Connectionist Temporal Classification (CTC) layer of the network to also give us a confidence score in addition to the predicted text. We then used this confidence score to bucket predictions in three ways:

1. If the confidence was high, we kept the prediction as is.
2. If the confidence was low, we simply filtered them out, making a bet that these were noise predictions.
3. If the confidence was somewhere in the middle, we then ran it through a lexicon generated from the Oxford English Dictionary, applying different transformations between and within word prediction boxes, attempting to combine words or split them in various ways to see if they were in the lexicon.

We also had to deal with issues caused by the previously mentioned fixed receptive image size of the Word Deep Net: namely, that a single “word” window might actually contain multiple words or only part of a very long word. We thus run these outputs along with the original outputs from the Word Detector through a module we call the Wordinator, which gives discrete bounding boxes for each individual OCRed word. This results in individual word coordinates along with their OCRed text.

For example, in the following debug visualization from our system you can see boxes around detected words before the Wordinator:



The Wordinator will break some of these boxes into individual word coordinate boxes, such as “of” and “Engineering”, which are currently part of the same box.

Finally, now that we had a fully working end-to-end system, we generated more than ten million synthetic words and trained our neural net for a very large number of iterations to squeeze out as much accuracy as we could. All of this finally gave us the accuracy, precision, and recall numbers that all met or exceeded the OCR state-of-the-art.

We briefly patted ourselves on the back, then began to prepare for the next tough stage: productionization.

Productionization

At this point, we had a collection of prototype Python and Lua scripts wrapping Torch — and a trained model, of course! — that showed that we could achieve state of the art OCR accuracy. However, this is a long way from a system an actual user can use in a distributed setting with reliability, performance, and solid engineering. We needed to create a distributed pipeline suitable for use by millions of users and a system replacing our prototype scripts. In addition, we had to do this without disrupting the existing OCR system using the commercial off the shelf SDK.

Here’ s a diagram of the productionized OCR pipeline:

Overall Productionized OCR Pipeline

We started by creating an abstraction for different OCR engines, including our own engine and the commercial one, and gated this using our in-house experiments framework, Stormcrow. This allowed us to introduce the skeleton of our new pipeline without disrupting the existing OCR system, which was already running in production for millions of our Business customers.

We also ported our Torch based model, including the CTC layer, to TensorFlow for a few reasons. First, we’ d already standardized on TensorFlow in production to make it easier to manage models and deployments. Second, we prefer to work with Python rather than Lua, and TensorFlow has excellent Python bindings.

In the new pipeline, mobile clients upload scanned document images to our in-house asynchronous work queue. When the upload is finished, we then send the image via a Remote Procedure Call (RPC) to a cluster of servers running the OCR service.

The actual OCR service uses OpenCV and TensorFlow, both written in C++ and with complicated library dependencies; so security exploits are a real concern. We’ve isolated the actual OCR portion into jails using technologies like LXC, CGroups, Linux Namespaces, and Seccomp to provide isolation and syscall whitelisting, using IPCs to talk into and out of the isolated container. If someone compromises the jail they will still be completely separated from the rest of our system.

Our jail infrastructure allows us to efficiently set up expensive resources a single time at startup, such as loading our trained models, then have these resources be cloned into a jail to satisfy a single OCR request. The resources are cloned Copy-on-Write into the forked jail and are read-only for how we use our models so it’s quite efficient and fast. We had to patch TensorFlow to make it easier to do this kind of forking. (We submitted the patch upstream.)

Once we get word bounding boxes and their OCRed text, we merge them back into the original PDF produced by the mobile document scanner as an OCR hidden layer. The user thus gets a PDF that has both the scanned image and the detected text. The OCRed text is also added to Dropbox’s search index. The user can now highlight and copy-paste text from the PDF, with the highlights going in the correct place due to our hidden word box coordinates. They can also search for the scanned PDF via its OCRed text on Dropbox.

Performance tuning

At this point, we now had an actual engineering pipeline (with unit tests and continual integration!), but still had performance issues.

The first question was whether we would use CPUs or GPUs in production at inference time. Training a deep net takes much longer than using it at inference time. It is common to use GPUs during training (as we did), as they vastly decrease the amount of time it takes to train a deep net. However, using GPUs at inference time is a harder call to make currently.

First, having high-end GPUs in a production data center such as Dropbox’ s is still a bit exotic and different than the rest of the fleet. In addition, GPU-based machines are more expensive and configurations are churning faster based on rapid development. We did an extensive analysis of how our Word Detector and Word Deep Net performed on CPUs vs GPUs, assuming full use of all cores on each CPU and the characteristics of the CPU. After much analysis, we decided that we could hit our performance targets on just CPUs at similar or lower costs than with GPU machines.

Once we decided on CPUs, we then needed to optimize our system BLAS libraries for the Word Deep Net, to tune our network a bit, and to configure TensorFlow to use available cores. Our Word Detector was also a significant bottleneck. We ended up essentially rewriting OpenCV’ s C++ MSER implementation in a more modular way to avoid duplicating slow work when doing two passes (to be able to handle both black on white text as well as white on black text); to expose more to our Python layer (the underlying MSER tree hierarchy) for more efficient processing; and to make the code actually readable. We also had to optimize the post-MSER Word Detection pipeline to tune and vectorize certain slow portions of it.

After all this work, we now had a productionized and highly-performant system that we could “shadow turn on” for a small number of users, leading us to the third phase: refinement.

Refinement

With our proposed system running silently in production side-by-side with the commercial OCR system, we needed to confirm that our system was truly better, as measured on real user data. We take user data privacy very seriously at Dropbox, so we couldn’ t just view and test random mobile document scanned images. Instead, we used the user-image donation flow detailed earlier to get evaluation images. We then used these donated images, being very careful about their privacy, to do a qualitative blackbox test of both OCR systems end-to-end, and were elated to find that we indeed performed the same or better than the older commercial OCR SDK, allowing us to ramp up our system to 100% of Dropbox Business users.

Next, we tested whether fine-tuning our trained deep net on these donated documents versus our hand chosen fine tuning image suite helped accuracy. Unfortunately, it didn’ t move the needle.

Another important refinement was doing orientation detection, which we had not done in the original pipeline. Images from the mobile document scanner can be rotated by 90° or even upside down. We built an orientation predictor using another deep net based on the Inception Resnet v2 architecture, changed the final layer to predict orientation, collected an orientation training and validation data set, and fine-tuned from an ImageNet-trained model biased towards our own needs. We put this orientation predictor into our pipeline, using its detected orientation to rotate the image to upright before doing word detection and OCRing.

One tricky aspect of the orientation predictor was that only a small percentage of images are actually rotated; we needed to make sure our system didn’ t inadvertently rotate upright images (the most common case) while trying to fix the orientation for the smaller number of non-upright images. In addition, we had to solve various tricky issues in combining our upright rotated images with the different ways the PDF file format can apply its own transformation matrices for rotation.

Finally, we were surprised to find some tough issues with the PDF file format containing our scanned OCRed hidden layer in Apple’ s native Preview application. Most PDF renderers respect spaces embedded in the text for copy and paste, but Apple’ s Preview application performs its own heuristics to determine word boundaries based on text position. This resulted in unacceptable quality for copy and paste from this PDF renderer, causing most spaces to be dropped and all the words to be “glommed together” . We had to do extensive testing across a range of PDF renderers to find the right PDF tricks and workarounds that would solve this problem.

In all, this entire round of researching, productionization, and refinement took about 8 months, at the end of which we had built and deployed a state-of-the-art OCR pipeline to millions of users using modern computer vision and deep neural network techniques. Our work also provides a solid foundation for future OCR-based products at Dropbox.

Interested in applying the latest machine learning research to hard, real–world problems and shipping to millions of Dropbox users? Our team is hiring!