Google Research Blog

The latest news from Research at Google

The Google Brain Team — Looking Back on 2017 (Part 2 of 2)

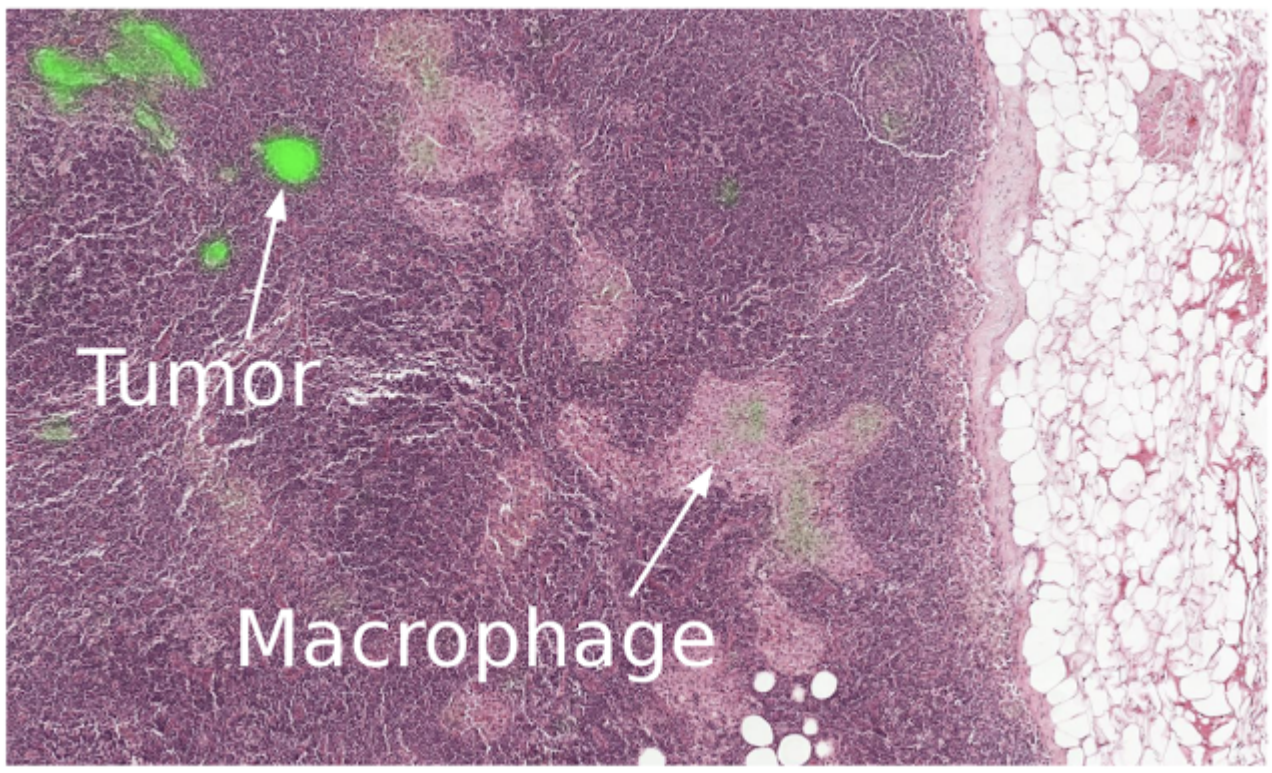
Friday, January 12, 2018

Posted by Jeff Dean, Google Senior Fellow, on behalf of the entire Google Brain Team

The Google Brain team works to advance the state of the art in artificial intelligence by research and systems engineering, as one part of the overall Google AI effort. In Part 1 of this blog post, we shared some of our work in 2017 related to our broader research, from designing new machine learning algorithms and techniques to understanding them, as well as sharing data, software, and hardware with the community. In this post, we'll dive into the research we do in some specific domains such as healthcare, robotics, creativity, fairness and inclusion, as well as share a little more about us.

Healthcare

We feel there is enormous potential for the application of machine learning techniques to healthcare. We are doing work across many different kinds of problems, including assisting pathologists in detecting cancer, understanding medical conversations to assist doctors and patients, and using machine learning to tackle a wide variety of problems in genomics, including an open-source release of a highly accurate variant calling system based on deep learning.



A lymph node biopsy, where our algorithm correctly identifies the tumor and not the benign macrophage.

We have continued our work on early detection of diabetic retinopathy (DR) and macular edema, building on the research paper we published December 2016 in the Journal of the American Medical Association (JAMA). In 2017, we moved this project from research project to actual clinical impact. We partnered with Verily (a life sciences company within Alphabet) to guide this work through the regulatory process, and together we are incorporating this technology into Nikon's line of Optos ophthalmology cameras. In addition, we are working to deploy this system in India, where there is a shortage of 127,000 eye doctors and as a result, almost half of patients are diagnosed too late — after the disease has already caused vision loss. As a part of a pilot, we've launched this system to help graders at Aravind Eye Hospitals to better diagnose diabetic eye disease. We are also working with our partners to understand the human factors affecting diabetic eye care, from ethnographic studies of patients and healthcare providers, to investigations on how eye care clinicians interact with the AI-enabled system.

First patient screened (top) and Iniya Paramasivam, a trained grader, viewing the output of the system (bottom).

We have also teamed up with researchers at leading healthcare organizations and medical centers including Stanford, UCSF, and University of Chicago to demonstrate the effectiveness of using machine learning to predict medical outcomes from de-identified medical records (i.e. given the current state of a patient, we believe we can predict the future for a patient by learning from millions of other patients' journeys, as a way of helping healthcare professionals make better decisions). We're very excited about this avenue of work and we look to forward to telling you more about it in 2018.

Robotics

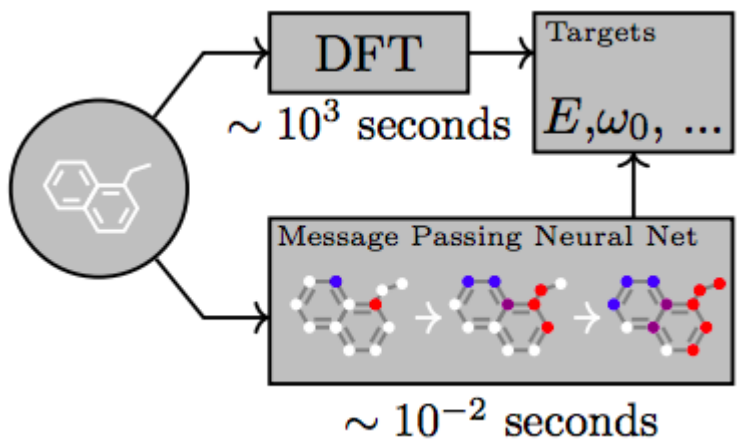
Our long-term goal in robotics is to design learning algorithms to allow robots to operate in messy, real-world environments and to quickly acquire new skills and capabilities via learning, rather than the carefully-controlled conditions and the small set of hand-programmed tasks that characterize today's robots. One thrust of our research is on developing techniques for physical robots to use their own experience and those of other robots to build new skills and capabilities, pooling the shared experiences in order to learn collectively. We are also exploring ways in which we can combine computer-based simulations of robotic tasks with physical robotic experience to learn new tasks more rapidly. While the physics of the simulator don't entirely match up with the real world, we have observed that for robotics, simulated experience plus a small amount of real-world experience gives significantly better results than even large amounts of real-world experience on its own.

In addition to real-world robotic experience and simulated robotic environments, we have developed robotic learning algorithms that can learn by observing human demonstrations of desired behaviors, and believe that this imitation learning approach is a highly promising way of imparting new abilities to robots very quickly, without explicit programming or even explicit specification of the goal of an activity. For example, below is a video of a robot learning to pour from a cup in just 15 minutes of real world experience by observing humans performing this task from different viewpoints and then trying to imitate the behavior. As we might be with our own three-year-old child, we're encouraged that it only spills a little!

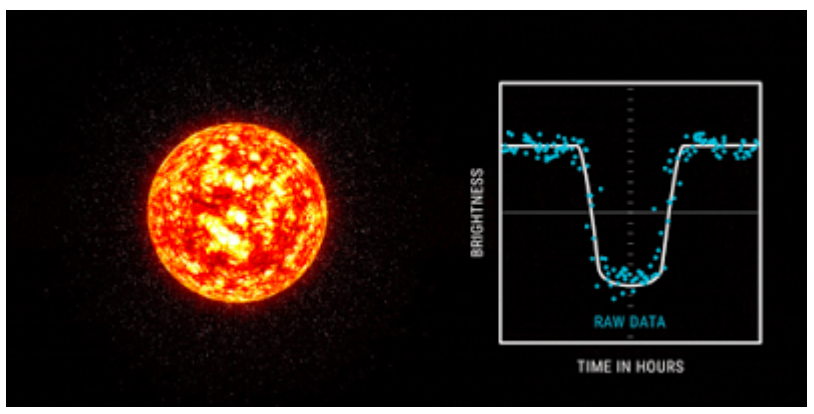
We also co-organized and hosted the first occurrence of the new Conference on Robot Learning (CoRL) in November to bring together researchers working at the intersection of machine learning and robotics. The summary of the event contains more information, and we look forward to next year's occurrence of the conference in Zürich.

Basic Science

We are also excited about the long term potential of using machine learning to help solve important problems in science. Last year, we utilized neural networks for predicting molecular properties in quantum chemistry, finding new exoplanets in astronomical datasets, earthquake aftershock prediction, and used deep learning to guide automated proof systems.



A Message Passing Neural Network predicts quantum properties of an organic molecule



Finding a new exoplanet: observing brightness of stars when planets block their light.

Creativity

We're very interested in how to leverage machine learning as a tool to assist people in creative endeavors. This year, we created an AI piano duet tool, helped YouTube musician Andrew Huang create new music (see also the behind the scenes video with Nat & Friends), and showed how to teach machines to draw.

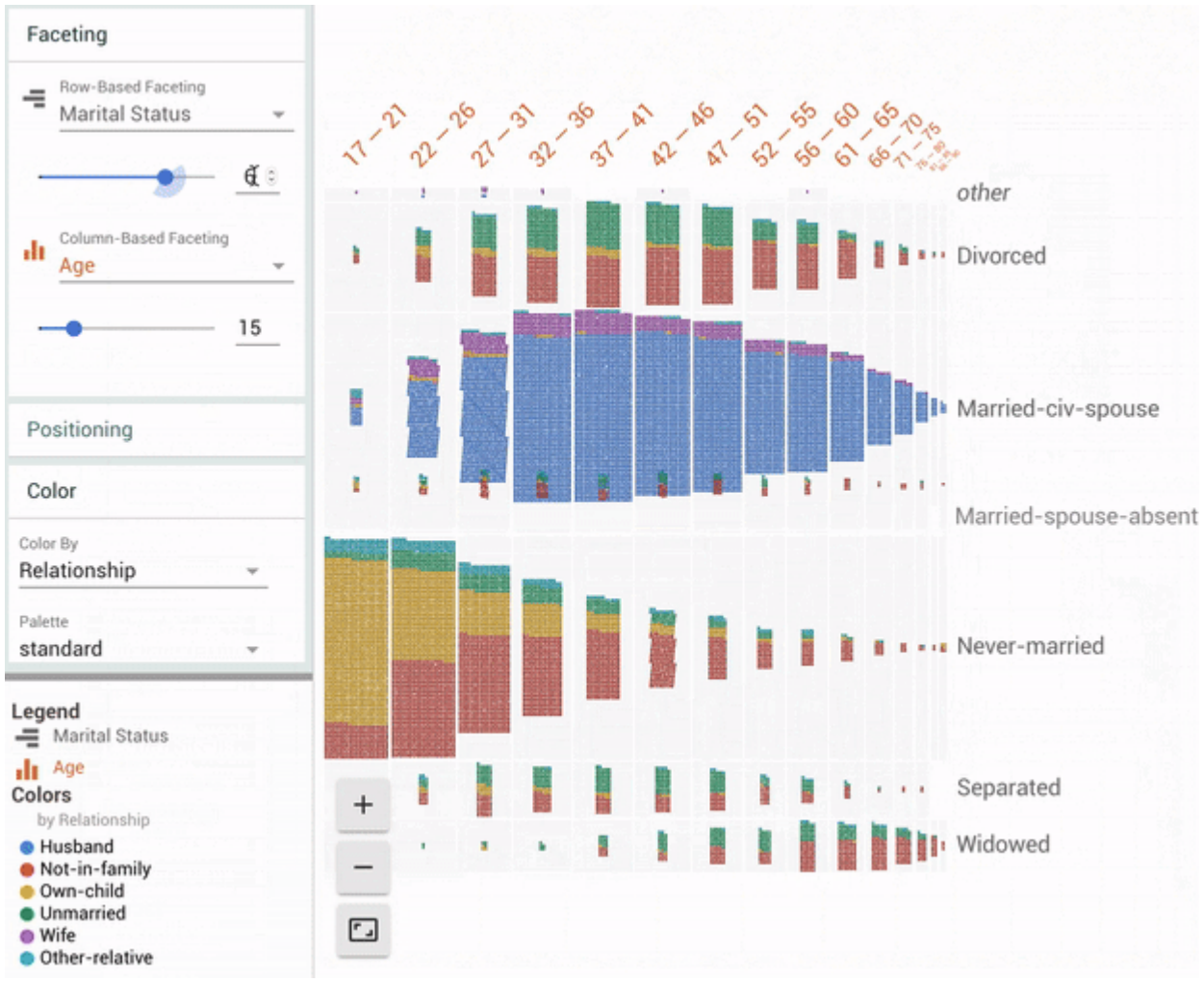
A garden drawn by the SketchRNN model; an interactive demo is available.

We also demonstrated how to control deep generative models running in the browser to create new music. This work won the NIPS 2017 Best Demo Award, making this the second year in a row that members of the Brain team's Magenta project have won this award, following on our receipt of the NIPS 2016 Best Demo Award for Interactive musical improvisation with Magenta. In the YouTube video below, you can listen to one part of the demo, the MusicVAE variational autoencoder model morphing smoothly from one melody to another.

People + AI Research (PAIR) Initiative

Advances in machine learning offer entirely new possibilities for how people might interact with computers. At the same time, it's critical to make sure that society can broadly benefit from the technology we're building. We see these opportunities and challenges as an urgent matter, and teamed up with a number of people throughout Google to create the People + AI Research (PAIR) initiative.

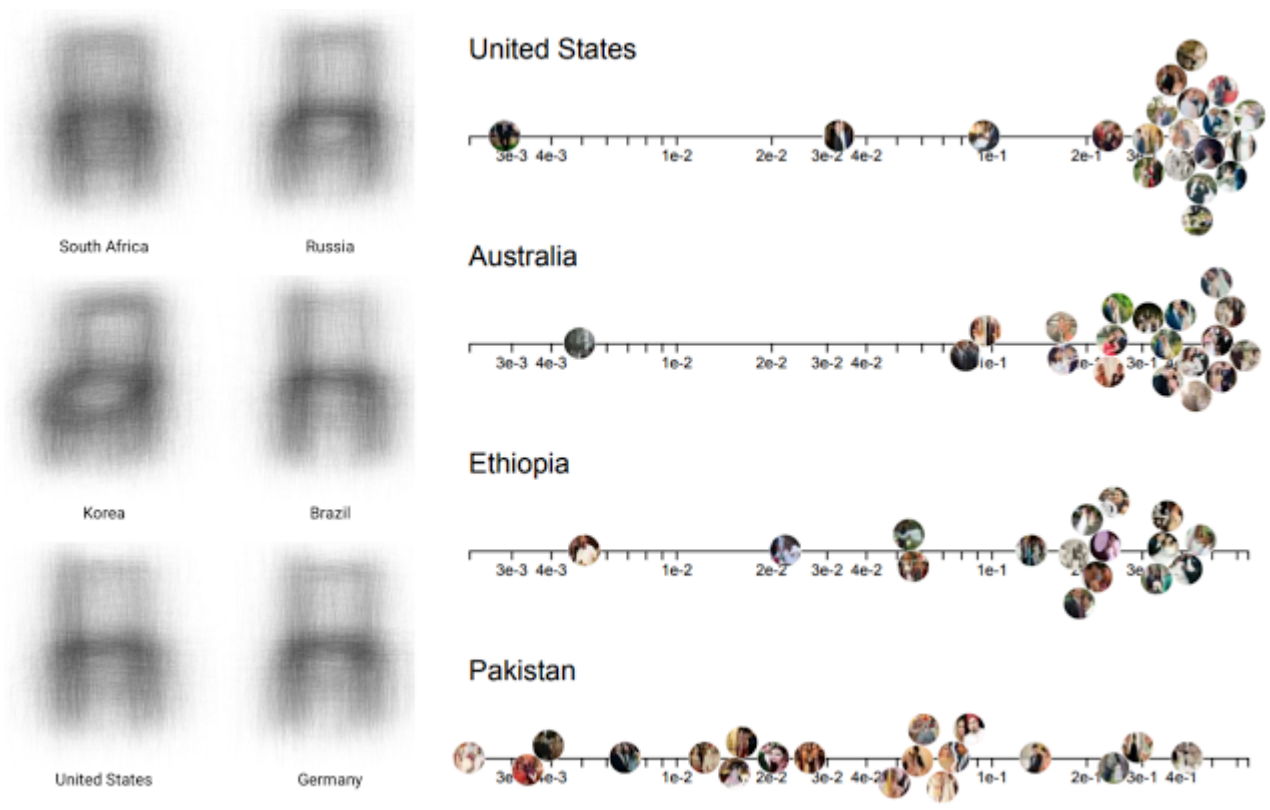
PAIR's goal is to study and design the most effective ways for people to interact with AI systems. We kicked off the initiative with a public symposium bringing together academics and practitioners across disciplines ranging from computer science, design, and even art. PAIR works on a wide range of topics, some of which we've already mentioned: helping researchers understand ML systems through work on interpretability and expanding the community of developers with deeplearn.js. Another example of our human-centered approach to ML engineering is the launch of Facets, a tool for visualizing and understanding training datasets.



Facets provides insights into your training datasets.

Fairness and Inclusion in Machine Learning

As ML plays an increasing role in technology, considerations of inclusivity and fairness grow in importance. The Brain team and PAIR have been working hard to make progress in these areas. We've published on how to avoid discrimination in ML systems via causal reasoning, the importance of geodiversity in open datasets, and posted an analysis of an open dataset to understand diversity and cultural differences. We've also been working closely with the Partnership on AI, a cross-industry initiative, to help make sure that fairness and inclusion are promoted as goals for all ML practitioners.



Cultural differences can surface in training data even in objects as "universal" as chairs, as observed in these doodle patterns on the left. The chart on the right shows how we uncovered geo-location biases in standard open source data sets such as ImageNet. Undetected or uncorrected, such biases may strongly influence model behavior.

We made this video in collaboration with our colleagues at Google Creative Lab as a non-technical introduction to some of the issues in this area.

Our Culture
One aspect of our group's research culture is to empower researchers and engineers to tackle the basic research problems that they view as most important. In September, we posted about our general approach to conducting research. Educating and mentoring young researchers is something we do through our research efforts. Our group hosted over 100 interns last year, and roughly 25% of our research publications in 2017 have intern co-authors. In 2016, we started the Google Brain Residency, a program for mentoring people who wanted to learn to do machine learning research. In the inaugural year (June 2016 to May 2017), 27 residents joined our group, and we posted updates about the first year of the program in halfway through and just after the end highlighting the research accomplishments of the residents. Many of the residents in the first year of the program have stayed on in our group as full-time researchers and research engineers, and most of those that did not have gone on to Ph.D. programs at top machine learning graduate programs like Berkeley, CMU, Stanford, NYU and Toronto. In July, 2017, we also welcomed our second cohort of 35 residents, who will be with us until July, 2018, and they've already done some exciting research and published at numerous research venues. We've now broadened the program to include many other research groups across Google and renamed it the Google AI Residency program (the application deadline for this year's program has just passed; look for information about next year's program at g.co/airesidency/apply).

Our work in 2017 spanned more than we've highlighted on in this two-part blog post. We believe in publishing our work in top research venues, and last year our group published 140 papers, including more than 60 at ICLR, ICML, and NIPS. To learn more about our work, you can peruse our research papers.

You can also meet some of our team members in this video, or read our responses to our second Ask Me Anything (AMA) post on r/MachineLearning (and check out the 2016's AMA, too).

The Google Brain team is becoming more spread out, with team members across North America and Europe. If the work we're doing sounds interesting and you'd like to join us, you can see our open positions and apply for internships, the AI Residency program, visiting faculty, or full-time research or engineering roles using the links at the bottom of g.co/brain. You can also follow our work throughout 2018 here on the Google Research blog, or on Twitter at @GoogleResearch. You can also follow my personal account at @JeffDean.

Thanks for reading!

33 条评论

Labels: Deep Learning , Google Brain , Research , TensorFlow

The Google Brain Team — Looking Back on 2017 (Part 1 of 2)

Thursday, January 11, 2018

Posted by Jeff Dean, Google Senior Fellow, on behalf of the entire Google Brain Team

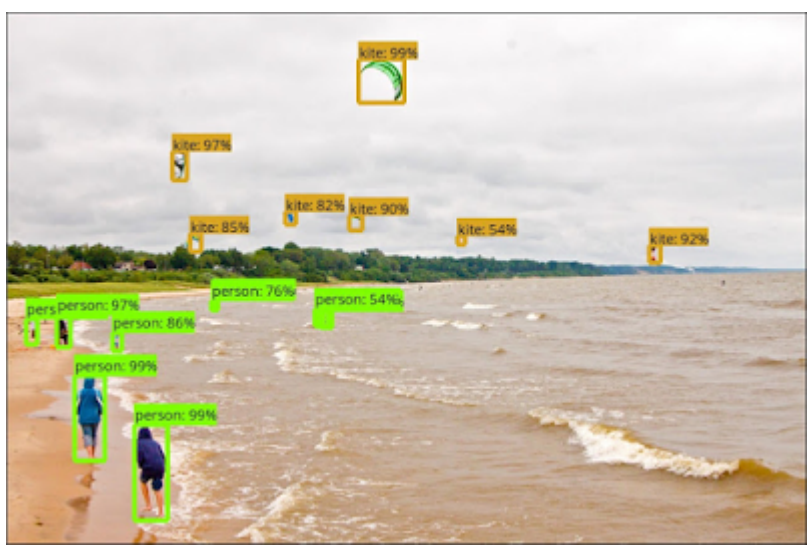
The Google Brain team works to advance the state of the art in artificial intelligence by research and systems engineering, as one part of the overall Google AI effort. Last year we shared a summary of our work in 2016. Since then, we've continued to make progress on our long-term research agenda of making machines intelligent, and have collaborated with a number of teams across Google and Alphabet to use the results of our research to improve people's lives. This first of two posts will highlight some of our work in 2017, including some of our basic research work, as well as updates on open source software, datasets, and new hardware for machine learning. In the second post we'll dive into the research we do in specific domains where machine learning can have a large impact, such as healthcare, robotics, and some areas of basic science, as well as

cover our work on creativity, fairness and inclusion and tell you a bit more about who we are.

Core Research
A significant focus of our team is pursuing research that advances our understanding and improves our ability to solve new problems in the field of machine learning. Below are several themes from our research last year.

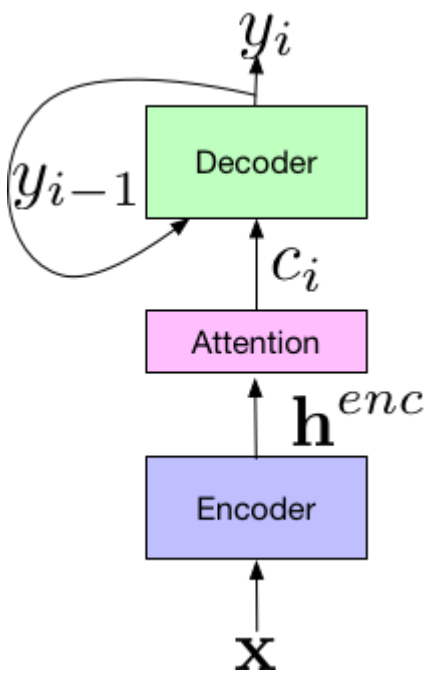
AutoML
The goal of automating machine learning is to develop techniques for computers to solve new machine learning problems automatically, without the need for human machine learning experts to intervene on every new problem. If we're ever going to have truly intelligent systems, this is a fundamental capability that we will need. We developed new approaches for designing neural network architectures using both reinforcement learning and evolutionary algorithms, scaled this work to state-of-the-art results on ImageNet classification and detection, and also showed how to learn new optimization algorithms and effective activation functions automatically. We are actively working with our Cloud AI team to bring this technology into the hands of Google customers, as well as continuing to push the research in many directions.

Convolutional architecture discovered by Neural Architecture Search



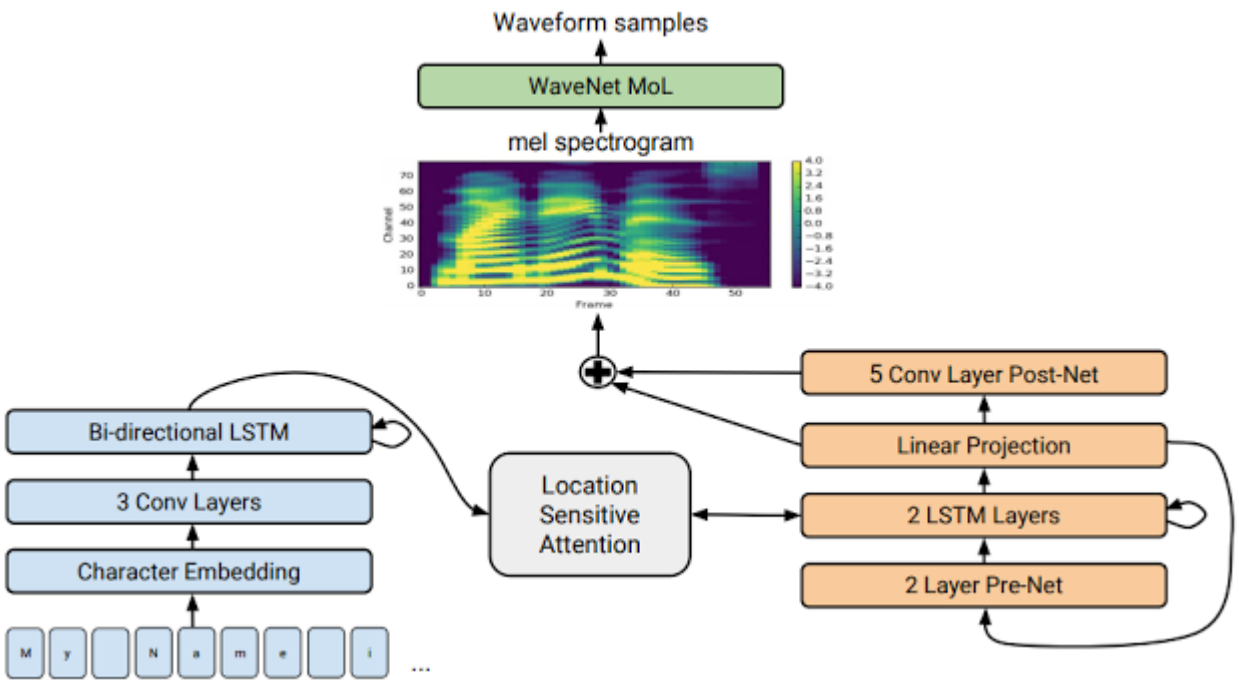
Object detection with a network discovered by AutoML

Speech Understanding and Generation
Another theme is on developing new techniques that improve the ability of our computing systems to understand and generate human speech, including our collaboration with the speech team at Google to develop a number of improvements for an end-to-end approach to speech recognition, which reduces the relative word error rate over Google's production speech recognition system by 16%. One nice aspect of this work is that it required many separate threads of research to come together (which you can find on Arxiv: 1, 2, 3, 4, 5, 6, 7, 8, 9).



Components of the Listen-Attend-Spell end-to-end model for speech recognition

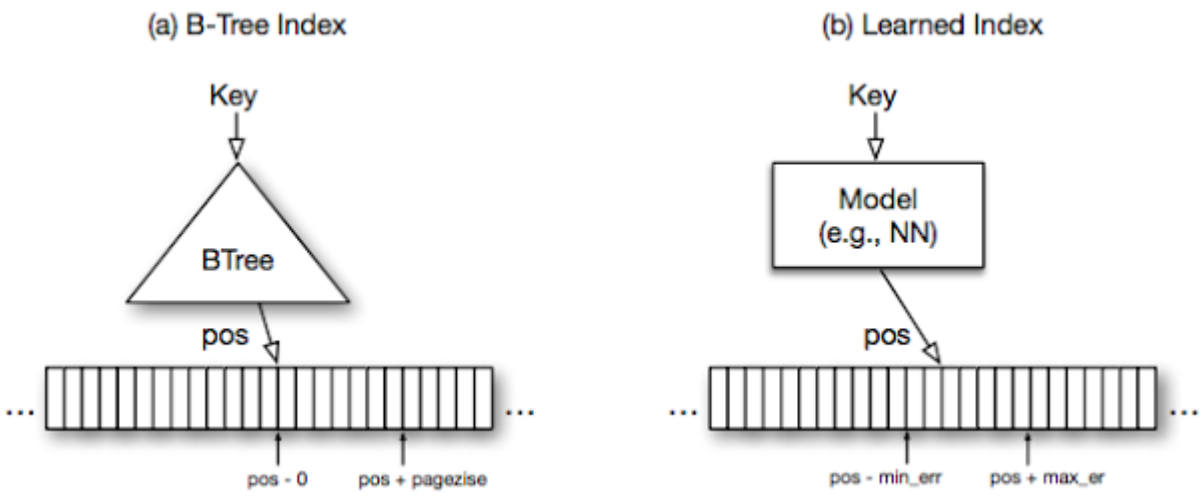
We also collaborated with our research colleagues on Google's Machine Perception team to develop a new approach for performing text-to-speech generation (Tacotron 2) that dramatically improves the quality of the generated speech. This model achieves a mean opinion score (MOS) of 4.53 compared to a MOS of 4.58 for professionally recorded speech like you might find in an audiobook, and 4.34 for the previous best computer-generated speech system. You can listen for yourself.



Tacotron 2's model architecture

New Machine Learning Algorithms and Approaches
We continued to develop novel machine learning algorithms and approaches, including work on capsules (which explicitly look for agreement in activated features as a way of evaluating many different noisy hypotheses when performing visual tasks), sparsely-gated mixtures of experts (which enable very large models that are still computational efficient), hypernetworks (which use the weights of one model to generate weights for another model), new kinds of multi-modal models (which perform multi-task learning across audio, visual, and textual inputs in the same model), attention-based mechanisms (as an alternative to convolutional and recurrent models), symbolic and non-symbolic learned optimization methods, a technique to back-propagate through discrete variables, and a few new reinforcement learning algorithmic improvements.

Machine Learning for Computer Systems
The use of machine learning to replace traditional heuristics in computer systems also greatly interests us. We have shown how to use reinforcement learning to make placement decisions for mapping computational graphs onto a set of computational devices that are better than human experts. With other colleagues in Google Research, we have shown in "The Case for Learned Index Structures" that neural networks can be both faster and much smaller than traditional data structures such as B-trees, hash tables, and Bloom filters. We believe that we are just scratching the surface in terms of the use of machine learning in core computer systems, as outlined in a NIPS workshop talk on Machine Learning for Systems and Systems for Machine Learning.



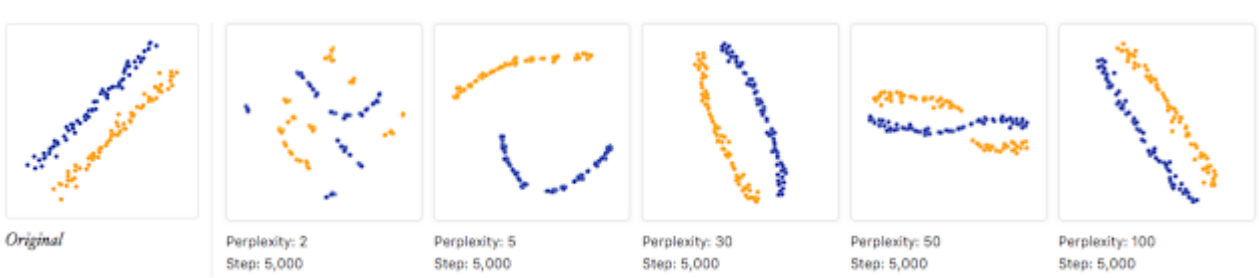
Learned Models as Index Structures

Privacy and Security
Machine learning and its interactions with security and privacy continue to be major research foci for us. We showed that machine learning techniques can be applied in a way that provides differential privacy guarantees, in a paper that received one of the best paper awards at ICLR 2017. We also continued our investigation into the properties of adversarial examples, including demonstrating adversarial examples in the physical world, and how to harness adversarial examples at scale during the training process to make models more robust to adversarial examples.

Understanding Machine Learning Systems
While we have seen impressive results with deep learning, it is important to understand why it works, and when it won't. In another one of the best paper awards of ICLR 2017, we showed that current machine learning theoretical frameworks fail to explain the impressive results of deep learning approaches. We also showed that the "flatness" of minima found by optimization methods is not as closely linked to good generalization as initially thought. In order to better understand how training proceeds in deep architectures, we published a series of papers analyzing random matrices, as they are the starting point of most training approaches. Another important avenue to understand deep learning is to better measure their performance. We showed the importance of good experimental design and statistical rigor in a recent study comparing many GAN approaches that found many popular enhancements to generative models do not actually improve performance. We hope this study will give an example for other researchers to follow in making robust experimental studies.

We are developing methods that allow better interpretability of machine learning systems. And in March, in collaboration with OpenAI, DeepMind, YC Research and others, we announced the launch of Distill, a new online open science journal dedicated to supporting human understanding of machine learning. It has gained a reputation for clear exposition of machine learning concepts and for excellent interactive visualization tools in its articles. In its first year, Distill has published many illuminating articles aimed at understanding the inner working of various machine learning techniques, and we look forward to the many more sure to come in 2018.

Feature Visualization

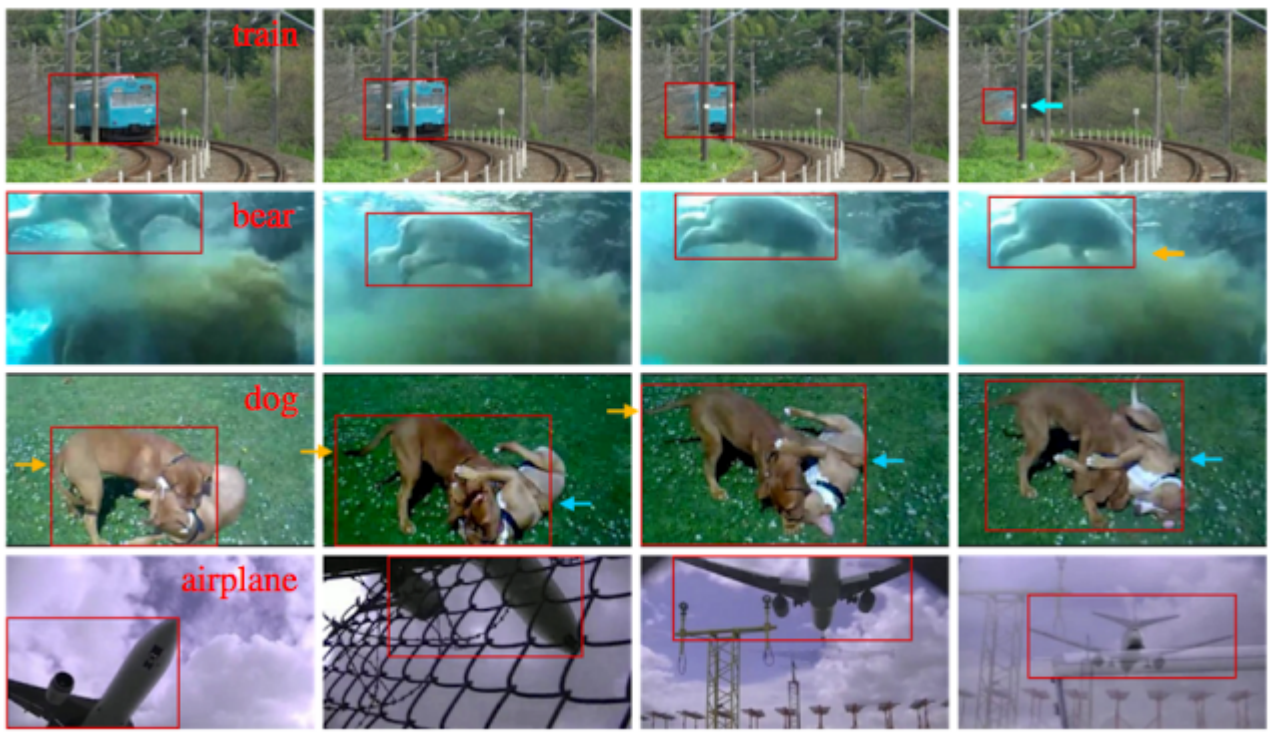


How to Use t-SNE effectively

Open Datasets for Machine Learning Research

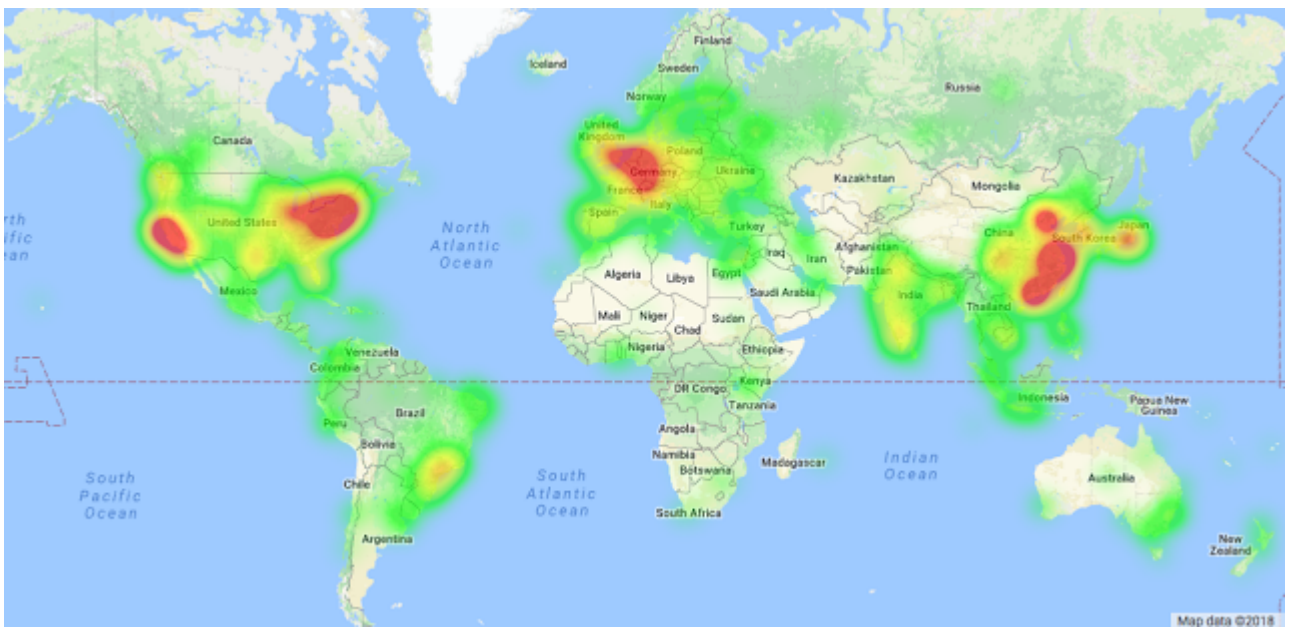
Open datasets like MNIST, CIFAR-10, ImageNet, SVHN, and WMT have pushed the field of machine learning forward tremendously. Our team and Google Research as a whole have been active in open-sourcing interesting new datasets for open machine learning research over the past year or so, by providing access to more large labeled datasets including:

- YouTube-8M: >7 million YouTube videos annotated with 4,716 different classes
- YouTube-Bounding Boxes: 5 million bounding boxes from 210,000 YouTube videos
- Speech Commands Dataset: thousands of speakers saying short command words
- AudioSet: 2 million 10-second YouTube clips labeled with 527 different sound events
- Atomic Visual Actions (AVA): 210,000 action labels across 57,000 video clips
- Open Images: 9M creative-commons licensed images labeled with 6000 classes
- Open Images with Bounding Boxes: 1.2M bounding boxes for 600 classes



Examples from the YouTube-Bounding Boxes dataset: Video segments sampled at 1 frame per second, with bounding boxes successfully identified around the items of interest.

TensorFlow and Open Source Software



A map showing the broad distribution of TensorFlow users (source)

Throughout our team's history, we have built tools that help us to conduct machine learning research and deploy machine learning systems in Google's many products. In November 2015, we open-sourced our second-generation machine learning framework, TensorFlow, with the hope of allowing the machine learning community as a whole to benefit from our investment in machine learning software tools. In February, we released TensorFlow 1.0, and in November, we released v1.4 with these significant additions: Eager execution for interactive imperative-style programming, XLA, an optimizing compiler for TensorFlow programs, and TensorFlow Lite, a lightweight solution for mobile and embedded devices. The pre-compiled TensorFlow binaries have now been downloaded more than 10 million times in over 180 countries, and the source code on GitHub now has more than 1,200 contributors.

In February, we hosted the first ever TensorFlow Developer Summit, with over 450 people attending live in Mountain View and more than 6,500 watching on live streams around the world, including at more than 85 local viewing events in 35 countries. All talks were recorded, with topics ranging from new features, techniques for using TensorFlow, or detailed looks under the hoods at low-level

TensorFlow abstractions. We'll be hosting another TensorFlow Developer Summit on March 30, 2018 in the Bay Area. Sign up now to save the date and stay updated on the latest news.

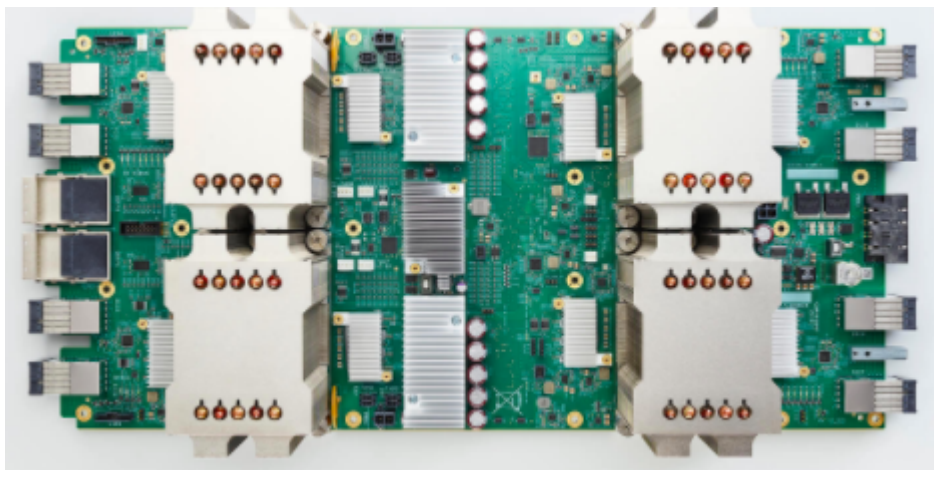
This rock-paper-scissors science experiment is a novel use of TensorFlow. We've been excited by the wide variety of uses of TensorFlow we saw in 2017, including automating cucumber sorting, finding sea cows in aerial imagery, sorting diced potatoes to make safer baby food, identifying skin cancer, helping to interpret bird call recordings in a New Zealand bird sanctuary, and identifying diseased plants in the most popular root crop on Earth in Tanzania!

In November, TensorFlow celebrated its second anniversary as an open-source project. It has been incredibly rewarding to see a vibrant community of TensorFlow developers and users emerge. TensorFlow is the #1 machine learning platform on GitHub and one of the top five repositories on GitHub overall, used by many companies and organizations, big and small, with more than 24,500 distinct repositories on GitHub related to TensorFlow. Many research papers are now published with open-source TensorFlow implementations to accompany the research results, enabling the community to more easily understand the exact methods used and to reproduce or extend the work.

TensorFlow has also benefited from other Google Research teams open-sourcing related work, including TF-GAN, a lightweight library for generative adversarial models in TensorFlow, TensorFlow Lattice, a set of estimators for working with lattice models, as well as the TensorFlow Object Detection API. The TensorFlow model repository continues to grow with an ever-widening set of models.

In addition to TensorFlow, we released deeplearn.js, an open-source hardware-accelerated implementation of deep learning APIs right in the browser (with no need to download or install anything). The deeplearn.js homepage has a number of great examples, including Teachable Machine, a computer vision model you train using your webcam, and Performance RNN, a real-time neural-network based piano composition and performance demonstration. We'll be working in 2018 to make it possible to deploy TensorFlow models directly into the deeplearn.js environment.

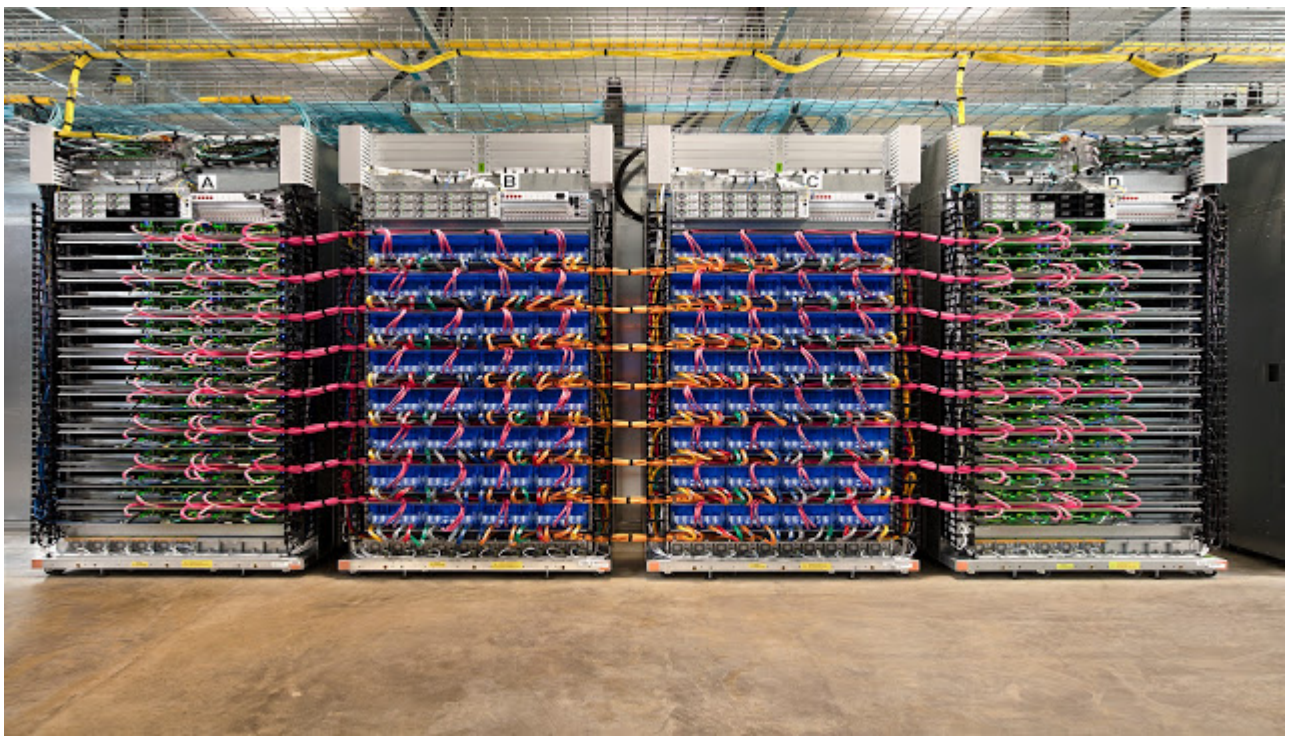
TPUs



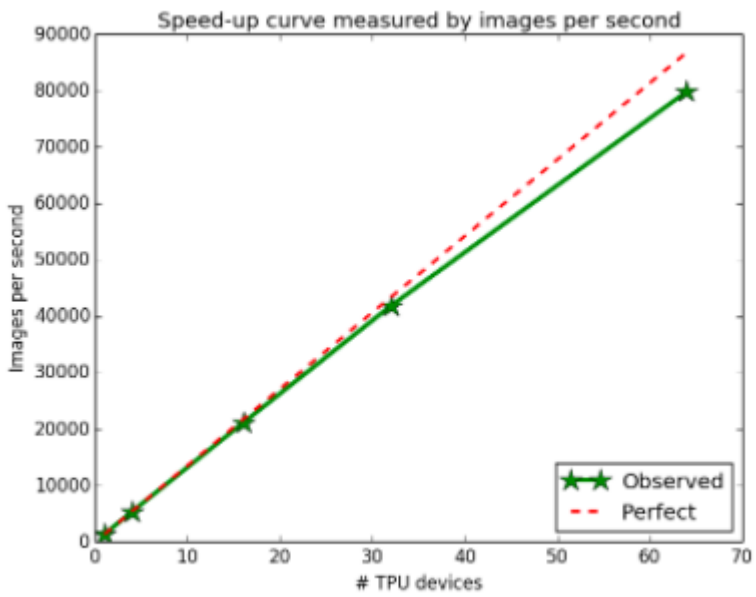
Cloud TPUs deliver up to 180 teraflops of machine learning acceleration

About five years ago, we recognized that deep learning would dramatically change the kinds of hardware we would need. Deep learning computations are very computationally intensive, but they have two special properties: they are largely composed of dense linear algebra operations (matrix multiples, vector operations, etc.), and they are very tolerant of reduced precision. We realized that we could take advantage of these two properties to build specialized hardware that can run neural network computations very efficiently. We provided design input to Google's Platforms team and they designed and produced our first generation Tensor Processing Unit (TPU): a single-chip ASIC designed to accelerate inference for deep learning models (inference is the use of an already-trained neural network, and is distinct from training). This first-generation TPU has been deployed in our data centers for three years, and it has been used to power deep learning models on every Google Search query, for Google Translate, for understanding images in Google Photos, for the

AlphaGo matches against Lee Sedol and Ke Jie, and for many other research and product uses. In June, we published a paper at ISCA 2017, showing that this first-generation TPU was 15X - 30X faster than its contemporary GPU or CPU counterparts, with performance/Watt about 30X - 80X better.



Cloud TPU Pods deliver up to 11.5 petaflops of machine learning acceleration



Experiments with ResNet-50 training on ImageNet show near-perfect speed-up as the number of TPU devices used increases.

Inference is important, but accelerating the training process is an even more important problem - and also much harder. The faster researchers can try a new idea, the more breakthroughs we can make. Our second-generation TPU, announced at Google I/O in May, is a whole system (custom ASIC chips, board and interconnect) that is designed to accelerate both training and inference, and we showed a single device configuration as well as a multi-rack deep learning supercomputer configuration called a TPU Pod. We announced that these second generation devices will be offered on the Google Cloud Platform as Cloud TPUs. We also unveiled the TensorFlow Research Cloud (TFRC), a program to provide top ML researchers who are committed to sharing their work with the world to access a cluster of 1,000 Cloud TPUs for free. In December, we presented work showing that we can train a ResNet-50 ImageNet model to a high level of accuracy in 22 minutes on a TPU Pod as compared to days or longer on a typical workstation. We think lowering research turnaround times in this fashion will dramatically increase the productivity of machine learning teams here at Google and at all of the organizations that use Cloud TPUs. If you're interested in Cloud TPUs, TPU Pods, or the TensorFlow Research Cloud, you can sign up to learn more at g.co/tpusignup. We're excited to enable many more engineers and researchers to use TPUs in 2018!

Thanks for reading!

(In part 2 we'll discuss our research in the application of machine learning to domains like healthcare, robotics, different fields of science, and creativity, as well as cover our work on fairness and inclusion.)

57 条评论

Labels: Deep Learning , Google Brain , Research , TensorFlow

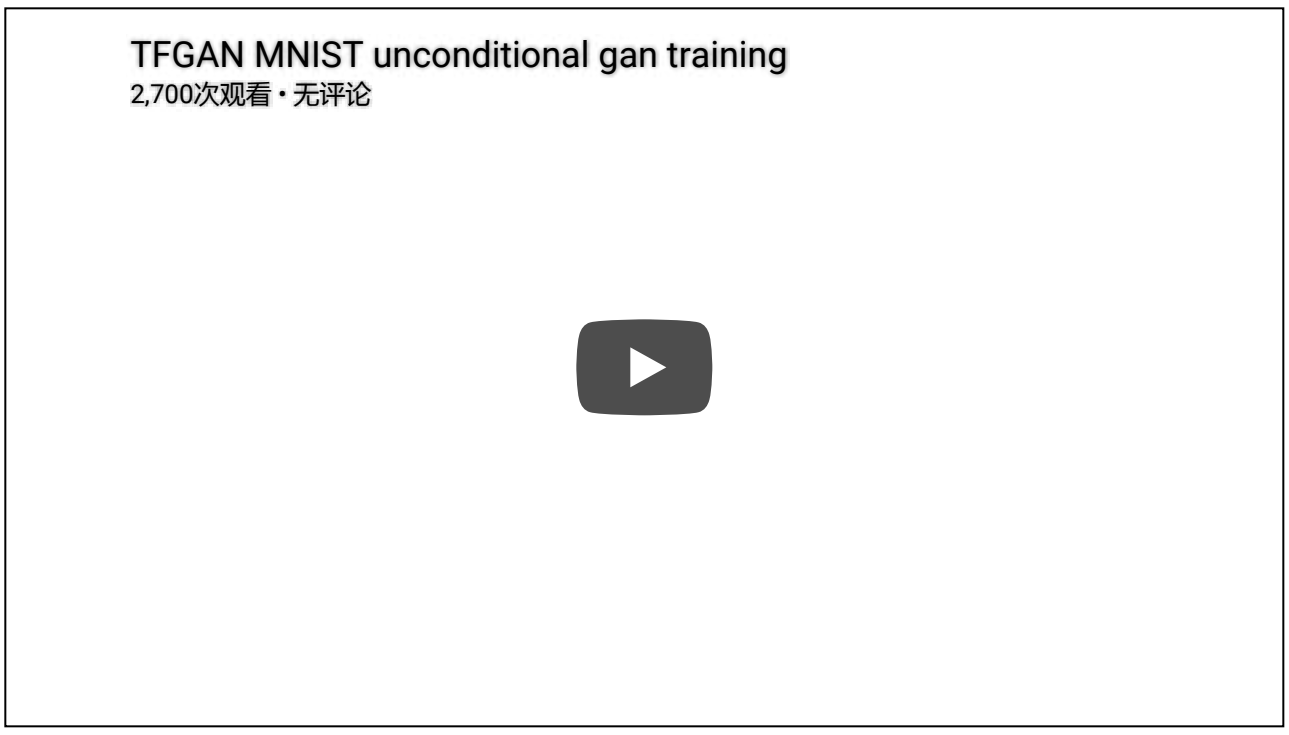
TFGAN: A Lightweight Library for Generative Adversarial Networks

Tuesday, December 12, 2017

Posted by Joel Shor, Senior Software Engineer, Machine Perception

(Crossposted on the Google Open Source Blog)

Training a neural network usually involves defining a loss function, which tells the network how close or far it is from its objective. For example, image classification networks are often given a loss function that penalizes them for giving wrong classifications; a network that mislabels a dog picture as a cat will get a high loss. However, not all problems have easily-defined loss functions, especially if they involve human perception, such as image compression or text-to-speech systems. Generative Adversarial Networks (GANs), a machine learning technique that has led to improvements in a wide range of applications including generating images from text, superresolution, and helping robots learn to grasp, offer a solution. However, GANs introduce new theoretical and software engineering challenges, and it can be difficult to keep up with the rapid pace of GAN research.

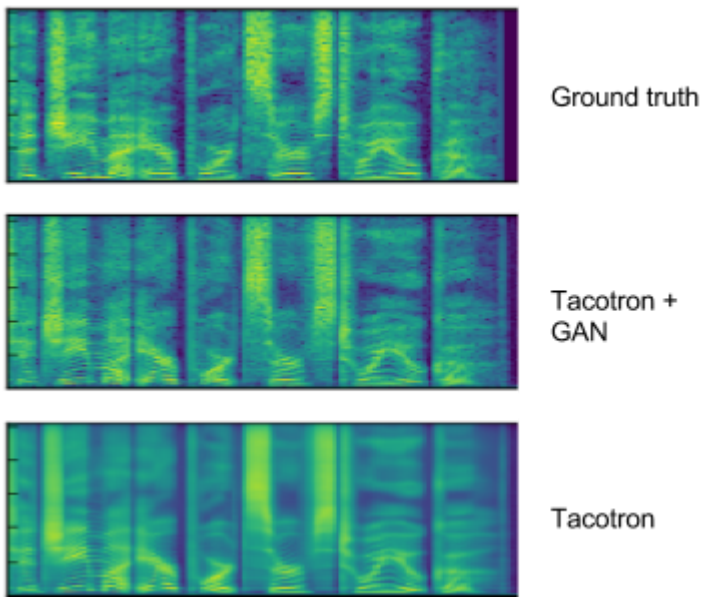


A video of a generator improving over time. It begins by producing random noise, and eventually learns to generate MNIST digits.

In order to make GANs easier to experiment with, we've open sourced TFGAN, a lightweight library designed to make it easy to train and evaluate GANs. It provides the infrastructure to easily train a GAN, provides well-tested loss and evaluation metrics, and gives easy-to-use examples that highlight the expressiveness and flexibility of TFGAN. We've also released a tutorial that includes a high-level API to quickly get a model trained on your data.

This demonstrates the effect of an adversarial loss on image compression. The top row shows image patches from the ImageNet dataset. The middle row shows the results of compressing and uncompressing an image through an image compression neural network trained on a traditional loss. The bottom row shows the results from a network trained with a traditional loss and an adversarial loss. The GAN-loss images are sharper and more detailed, even if they are less like the original.

TFGAN supports experiments in a few important ways. It provides simple function calls that cover the majority of GAN use-cases so you can get a model running on your data in just a few lines of code, but is built in a modular way to cover more exotic GAN designs as well. You can just use the modules you want — loss, evaluation, features, training, etc. are all independent. TFGAN's lightweight design also means you can use it alongside other frameworks, or with native TensorFlow code. GAN models written using TFGAN will easily benefit from future infrastructure improvements, and you can select from a large number of already-implemented losses and features without having to rewrite your own. Lastly, the code is well-tested, so you don't have to worry about numerical or statistical mistakes that are easily made with GAN libraries.



Most neural text-to-speech (TTS) systems produce over-smoothed spectrograms. When applied to the Tacotron TTS system, a GAN can recreate some of the realistic-texture, which reduces artifacts in the resulting audio.

When you use TFGAN, you'll be using the same infrastructure that many Google researchers use, and you'll have access to the cutting-edge improvements that we develop with the library. Anyone can contribute to the github repositories, which we hope will facilitate code-sharing among ML researchers and users.

41 条评论

Labels: Machine Learning , open source , Software , TensorFlow

On-Device Conversational Modeling with TensorFlow Lite

Tuesday, November 14, 2017

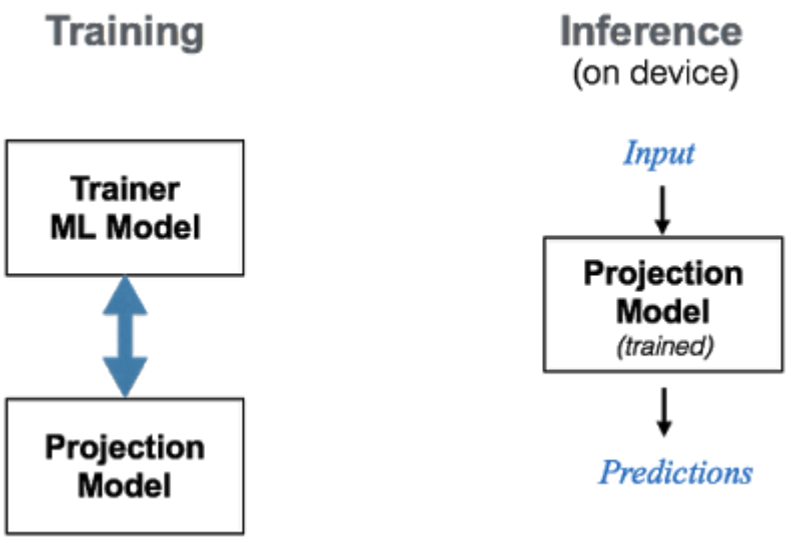
Posted by Sujith Ravi, Research Scientist, Google Expander Team

Earlier this year, we launched Android Wear 2.0 which featured the first "on-device" machine learning technology for smart messaging. This enabled cloud-based technologies like Smart Reply, previously available in Gmail, Inbox and Allo, to be used directly within any application for the first time, including third-party messaging apps, without ever having to connect to the cloud. So you can respond to incoming chat messages on the go, directly from your smartwatch.

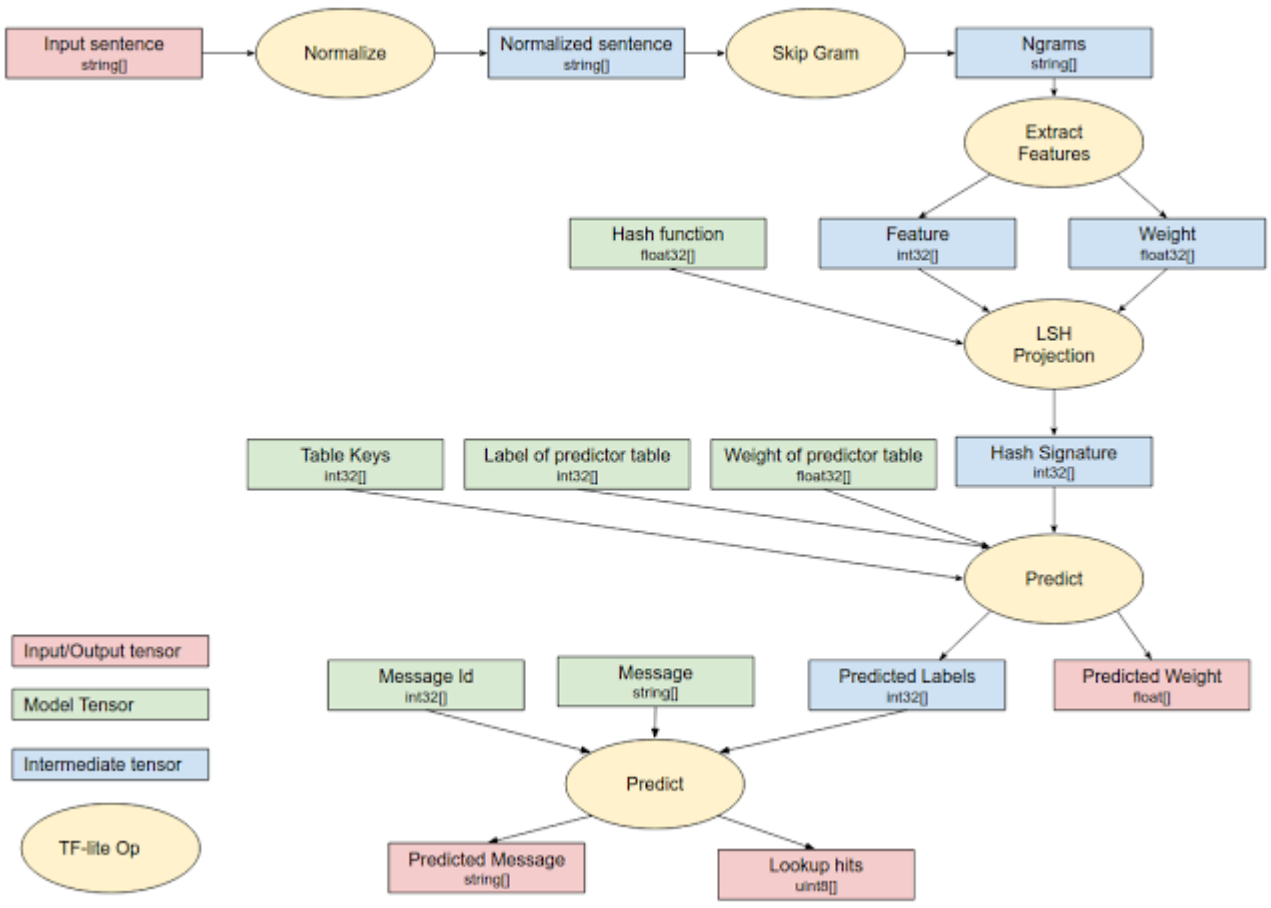
Today, we announce TensorFlow Lite, TensorFlow's lightweight solution for mobile and embedded devices. This framework is optimized for low-latency inference of machine learning models, with a focus on small memory footprint and fast performance. As part of the library, we have also released an on-device conversational model and a demo app that provides an example of a natural language application powered by TensorFlow Lite, in order to make it easier for developers and researchers to build new machine intelligence features powered by on-device inference. This model generates reply suggestions to input conversational chat messages, with efficient inference that can be easily plugged in to your chat application to power on-device conversational intelligence.

The on-device conversational model we have released uses a new ML architecture for training compact neural networks (as well as other machine learning models) based on a joint optimization framework, originally presented in ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections. This architecture can run efficiently on mobile devices with limited computing power and memory, by using efficient "projection" operations that transform any input to a compact bit vector representation — similar inputs are projected to nearby vectors that are dense or sparse depending on type of projection. For example, the messages *"hey, how's it going?"* and *"How's it going buddy?"*, might be projected to the same vector representation.

Using this idea, the conversational model combines these efficient operations at low computation and memory footprint. We trained this on-device model end-to-end using an ML framework that jointly trains two types of models — a compact *projection* model (as described above) combined with a *trainer* model. The two models are trained in a joint fashion, where the projection model learns from the trainer model — the trainer is characteristic of an expert and modeled using larger and more complex ML architectures, whereas the projection model resembles a student that learns from the expert. During training, we can also stack other techniques such as quantization or distillation to achieve further compression or selectively optimize certain portions of the objective function. Once trained, the smaller projection model is able to be used directly for inference on device.



For inference, the trained projection model is compiled into a set of TensorFlow Lite operations that have been optimized for fast execution on mobile platforms and executed directly on device. The TensorFlow Lite inference graph for the on-device conversational model is shown here.



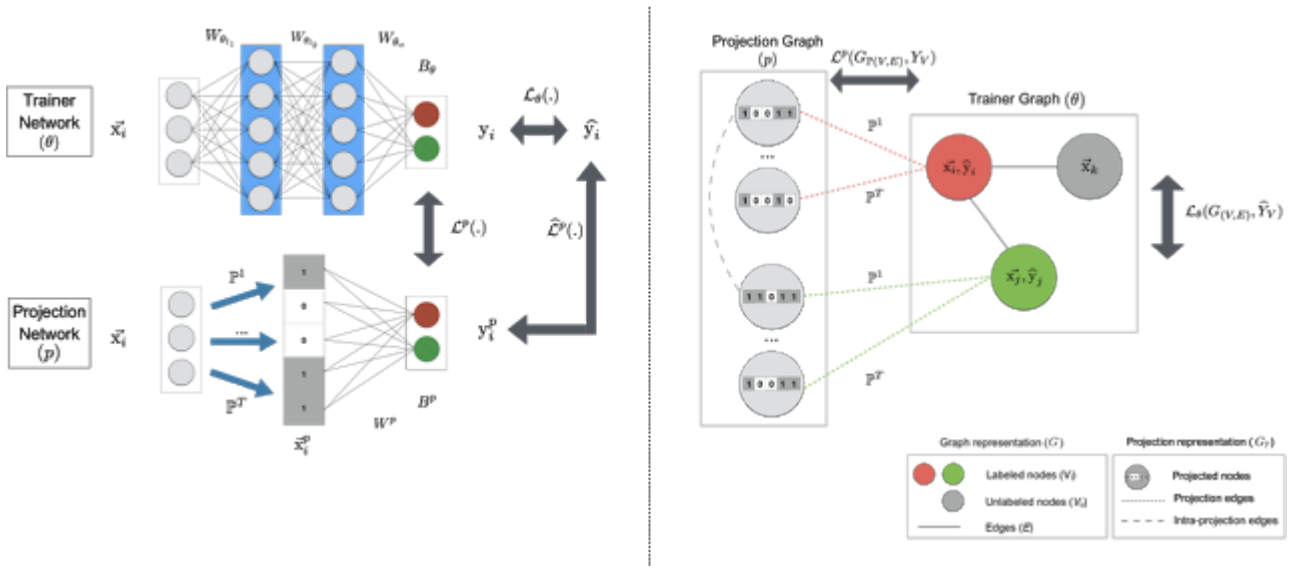
TensorFlow Lite execution for the On-Device Conversational Model.

The open-source conversational model released today (along with code) was trained end-to-end using the joint ML architecture described above. Today's release also includes a demo app, so you can easily download and try out one-touch smart replies on your mobile device. The architecture enables easy configuration for model size and prediction quality based on application needs. You can find a list of sample messages where this model does well here. The system can also fall back to suggesting replies from a fixed set that was learned and compiled from popular response intents observed in chat conversations. The underlying model is different from the ones Google uses for Smart Reply responses in its apps[1].

Beyond Conversational Models

Interestingly, the ML architecture described above permits flexible choices for the underlying model. We also designed the architecture to be compatible with different machine learning approaches — for example, when used with TensorFlow deep learning, we learn a lightweight neural network (*ProjectionNet*) for the underlying model, whereas a different architecture (*ProjectionGraph*) represents the model using a graph framework instead of a neural network.

The joint framework can also be used to train lightweight on-device models for other tasks using different ML modeling architectures. As an example, we derived a ProjectionNet architecture that uses a complex feed-forward or recurrent architecture (like LSTM) for the trainer model coupled with a simple projection architecture comprised of dynamic projection operations and a few, narrow fully-connected layers. The whole architecture is trained end-to-end using backpropagation in TensorFlow and once trained, the compact ProjectionNet is directly used for inference. Using this method, we have successfully trained tiny ProjectionNet models that achieve significant reduction in model sizes (up to several orders of magnitude reduction) and high performance with respect to accuracy on multiple visual and language classification tasks (a few examples here). Similarly, we trained other lightweight models using our graph learning framework, even in semi-supervised settings.



ML architecture for training on-device models: **ProjectionNet** trained using deep learning (left), and **ProjectionGraph** trained using graph learning (right).

We will continue to improve and release updated TensorFlow Lite models in open-source. We think that the released model (as well as future models) learned using these ML architectures may be reused for many natural language and computer vision applications or plugged into existing apps for enabling machine intelligence. We hope that the machine learning and natural language processing communities will be able to build on these to address new problems and use-cases we have not yet conceived.

Acknowledgments
Yicheng Fan and Gaurav Nemade contributed immensely to this effort. Special thanks to Rajat Monga, Andre Hentz, Andrew Selle, Sarah Sirajuddin, and Anitha Vijayakumar from the TensorFlow team; Robin Dua, Patrick McGregor, Andrei Broder, Andrew Tomkins and the Google Expander team.

---

1 The released on-device model was trained to optimize for small size and low latency applications on mobile phones and wearables. Smart Reply predictions in Google apps, however are generated using larger, more complex models. In production systems, we also use multiple classifiers that are trained to detect inappropriate content and apply further filtering and tuning to optimize user experience and quality levels. We recommend that developers using the open-source TensorFlow Lite version also follow such practices for their end applications. ↩

Labels: Expander , Machine Learning , On-device Learning , open source , TensorFlow

Latest Innovations in TensorFlow Serving

Thursday, November 02, 2017

Posted by Chris Olston, Research Scientist, and Noah Fiedel, Software Engineer, TensorFlow Serving

Since initially open-sourcing TensorFlow Serving in February 2016, we've made some major enhancements. Let's take a look back at where we started, review our progress, and share where we are headed next.

Before TensorFlow Serving, users of TensorFlow inside Google had to create their own serving system from scratch. Although serving might appear easy at first, one-off serving solutions quickly grow in complexity. Machine Learning (ML) serving systems need to support model versioning (for model updates with a rollback option) and multiple models (for experimentation via A/B testing), while ensuring that concurrent models achieve high throughput on hardware accelerators (GPUs and TPUs) with low latency. So we set out to create a single, general TensorFlow Serving software stack.

We decided to make it open-sourceable from the get-go, and development started in September 2015. Within a few months, we created the initial end-to-end working system and our open-source release in February 2016.

Over the past year and half, with the help of our users and partners inside and outside our company, TensorFlow Serving has advanced performance, best practices, and standards:

- **Out-of-the-box optimized serving and customizability**: We now offer a pre-built canonical serving binary, optimized for modern CPUs with AVX, so developers don't need to assemble their own binary from our libraries unless they have exotic needs. At the same time, we added a registry-based framework, allowing our libraries to be used for custom (or even non-TensorFlow) serving scenarios.

- **Multi-model serving**: Going from one model to multiple concurrently-served models presents several performance obstacles. We serve multiple models smoothly by (1) loading in isolated thread pools to avoid incurring latency spikes on other models taking traffic; (2) accelerating initial loading of all models in parallel upon server start-up; (3) multi-model batch interleaving to multiplex hardware accelerators (GPUs/TPUs).
- **Standardized model format**: We added SavedModel to TensorFlow 1.0, giving the community a single standard model format that works across training and serving.
- **Easy-to-use inference APIs**: We released easy-to-use APIs for common inference tasks (classification, regression) that we know work for a wide swathe of our applications. To support more advanced use-cases we support a lower-level tensor-based API (predict) and a new multi-inference API that enables multi-task modeling.

All of our work has been informed by close collaborations with: (a) Google's ML SRE team, which helps ensure we are robust and meet internal SLAs; (b) other Google machine learning infrastructure teams including ads serving and TFX; (c) application teams such as Google Play; (d) our partners at the UC Berkeley RISE Lab, who explore complementary research problems with the Clipper serving system; (e) our open-source user base and contributors.

TensorFlow Serving is currently handling tens of millions of inferences per second for 1100+ of our own projects including Google's Cloud ML Prediction. Our core serving code is available to all via our open-source releases.

Looking forward, our work is far from done and we are exploring several avenues of innovation. Today we are excited to share early progress in two experimental areas:

- **Granular batching**: A key technique we employ to achieve high throughput on specialized hardware (GPUs and TPUs) is "batching": processing multiple examples jointly for efficiency. We are developing technology and best practices to improve batching to: (a) enable batching to target just the GPU/TPU portion of the computation, for maximum efficiency; (b) enable batching within recursive neural networks, used to process sequence data e.g. text and event sequences. We are experimenting with batching arbitrary sub-graphs using the Batch/Unbatch op pair.
- **Distributed model serving**: We are looking at model sharding techniques as a means of handling models that are too large to fit on one server node or sharing sub-models in a memory-efficient way. We recently launched a 1TB+ model in production with good results, and hope to open-source this capability soon.

Thanks again to all of our users and partners who have contributed feedback, code and ideas. Join the project at: github.com/tensorflow/serving.

33 条评论

Labels: Google Brain , Machine Learning , open source , TensorFlow

Eager Execution: An imperative, define-by-run interface to TensorFlow

Tuesday, October 31, 2017

Posted by Asim Shankar and Wolff Dobson, Google Brain Team

Today, we introduce eager execution for TensorFlow. Eager execution is an imperative, define-by-run interface where operations are executed immediately as they are called from Python. This makes it easier to get started with TensorFlow, and can make research and development more intuitive.

The benefits of eager execution include:

- Fast debugging with immediate run-time errors and integration with Python tools
- Support for dynamic models using easy-to-use Python control flow
- Strong support for custom and higher-order gradients
- Almost all of the available TensorFlow operations

Eager execution is available now as an experimental feature, so we're looking for feedback from the community to guide our direction.

To understand this all better, let's look at some code. This gets pretty technical; familiarity with TensorFlow will help.

Using Eager Execution

When you enable eager execution, operations execute immediately and return their values to Python without requiring a `Session.run()`. For example, to multiply two matrices together, we write this:

```
import tensorflow as tf
import tensorflow.contrib.eager as tfe

tfe.enable_eager_execution()

x = [[2.]]
m = tf.matmul(x, x)
```

It's straightforward to inspect intermediate results with print or the Python debugger.

```
print(m)
# The 1x1 matrix [[4.]]
```

Dynamic models can be built with Python flow control. Here's an example of the Collatz conjecture using TensorFlow's arithmetic operations:

```
a = tf.constant(12)
counter = 0
while not tf.equal(a, 1):
  if tf.equal(a % 2, 0):
    a = a / 2
  else:
    a = 3 * a + 1
  print(a)
```

Here, the use of the `tf.constant(12)` Tensor object will promote all math operations to tensor operations, and as such all return values with be tensors.

Gradients

Most TensorFlow users are interested in automatic differentiation. Because different operations can occur during each call, we record all forward operations to a tape, which is then played backwards when computing gradients. After we've computed the gradients, we discard the tape.

If you're familiar with the `autograd` package, the API is very similar. For example:

```
def square(x):
  return tf.multiply(x, x)

grad = tfe.gradients_function(square)

print(square(3.))    # [9.]
print(grad(3.))      # [6.]
```

The `gradients_function` call takes a Python function `square()` as an argument and returns a Python callable that computes the partial derivatives of `square()` with respect to its inputs. So, to get the derivative of `square()` at 3.0, invoke `grad(3.0)`, which is 6.

The same `gradients_function` call can be used to get the second derivative of square:

```
gradgrad = tfe.gradients_function(lambda x: grad(x)[0])

print(gradgrad(3.))  # [2.]
```

As we noted, control flow can cause different operations to run, such as in this example.

```
def abs(x):
  return x if x > 0. else -x

grad = tfe.gradients_function(abs)

print(grad(2.0))  # [1.]
print(grad(-2.0)) # [-1.]
```

Custom Gradients

Users may want to define custom gradients for an operation, or for a function. This may be useful for multiple reasons, including providing a more efficient or more numerically stable gradient for a sequence of operations.

Here is an example that illustrates the use of custom gradients. Let's start by looking at the function $log(1 + e^x)$, which commonly occurs in the computation of cross entropy and log likelihoods.

```
def log1pexp(x):
  return tf.log(1 + tf.exp(x))
```

```
grad_log1pexp = tfe.gradients_function(log1pexp)

# The gradient computation works fine at x = 0.
print(grad_log1pexp(0.))
# [0.5]
# However it returns a `nan` at x = 100 due to numerical instability.
print(grad_log1pexp(100.))
# [nan]
```

We can use a custom gradient for the above function that analytically simplifies the gradient expression. Notice how the gradient function implementation below reuses an expression (`tf.exp(x)`) that was computed during the forward pass, making the gradient computation more efficient by avoiding redundant computation.

```
@tfe.custom_gradient
def log1pexp(x):
  e = tf.exp(x)
  def grad(dy):
    return dy * (1 - 1 / (1 + e))
  return tf.log(1 + e), grad
grad_log1pexp = tfe.gradients_function(log1pexp)

# Gradient at x = 0 works as before.
print(grad_log1pexp(0.))
# [0.5]
# And now gradient computation at x=100 works as well.
print(grad_log1pexp(100.))
# [1.0]
```

Building models

Models can be organized in classes. Here's a model class that creates a (simple) two layer network that can classify the standard MNIST handwritten digits.

```
class MNISTModel(tfe.Network):
  def __init__(self):
    super(MNISTModel, self).__init__()
    self.layer1 = self.track_layer(tf.layers.Dense(units=10))
    self.layer2 = self.track_layer(tf.layers.Dense(units=10))
  def call(self, input):
    """Actually runs the model."""
    result = self.layer1(input)
    result = self.layer2(result)
    return result
```

We recommend using the classes (not the functions) in tf.layers since they create and contain model parameters (variables). Variable lifetimes are tied to the lifetime of the layer objects, so be sure to keep track of them.

Why are we using `tfe.Network`? A Network is a container for layers and is a `tf.layer.Layer` itself, allowing `Network` objects to be embedded in other `Network` objects. It also contains utilities to assist with inspection, saving, and restoring.

Even without training the model, we can imperatively call it and inspect the output:

```
# Let's make up a blank input image
model = MNISTModel()
batch = tf.zeros([1, 1, 784])
print(batch.shape)
# (1, 1, 784)
result = model(batch)
print(result)
# tf.Tensor([[[ 0.  0., ...., 0.]]], shape=(1, 1, 10), dtype=float32)
```

Note that we do not need any placeholders or sessions. The first time we pass in the input, the sizes of the layers' parameters are set.

To train any model, we define a loss function to optimize, calculate gradients, and use an optimizer to update the variables. First, here's a loss function:

```
def loss_function(model, x, y):
  y_ = model(x)
  return tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=y_)
```

And then, our training loop:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001)
for (x, y) in tfe.Iterator(dataset):
```

```
  grads = tfe.implicit_gradients(loss_function)(model, x, y)
  optimizer.apply_gradients(grads)
```

`implicit_gradients()` calculates the derivatives of `loss_function` with respect to all the TensorFlow variables used during its computation.

We can move computation to a GPU the same way we've always done with TensorFlow:

```
with tf.device("/gpu:0"):
  for (x, y) in tfe.Iterator(dataset):
    optimizer.minimize(lambda: loss_function(model, x, y))
```

(Note: We're shortcutting storing our loss and directly calling the `optimizer.minimize`, but you could also use the `apply_gradients()` method above; they are equivalent.)

Using Eager with Graphs

Eager execution makes development and debugging far more interactive, but TensorFlow graphs have a lot of advantages with respect to distributed training, performance optimizations, and production deployment.

The same code that executes operations when eager execution is enabled will construct a graph describing the computation when it is not. To convert your models to graphs, simply run the same code in a new Python session where eager execution hasn't been enabled, as seen, for example, in the MNIST example. The value of model variables can be saved and restored from checkpoints, allowing us to move between eager (imperative) and graph (declarative) programming easily. With this, models developed with eager execution enabled can be easily exported for production deployment.

In the near future, we will provide utilities to selectively convert portions of your model to graphs. In this way, you can fuse parts of your computation (such as internals of a custom RNN cell) for high-performance, but also keep the flexibility and readability of eager execution.

How does my code change?

Using eager execution should be intuitive to current TensorFlow users. There are only a handful of eager-specific APIs; most of the existing APIs and operations work with eager enabled. Some notes to keep in mind:

- As with TensorFlow generally, we recommend that if you have not yet switched from queues to using `tf.data` for input processing, you should. It's easier to use and usually faster. For help, see this blog post and the documentation page.
- Use object-oriented layers, like `tf.layer.Conv2D()` or Keras layers; these have explicit storage for variables.
- For most models, you can write code so that it will work the same for both eager execution and graph construction. There are some exceptions, such as dynamic models that use Python control flow to alter the computation based on inputs.
- Once you invoke `tfe.enable_eager_execution()`, it cannot be turned off. To get graph behavior, start a new Python session.

Getting started and the future

This is still a preview release, so you may hit some rough edges. To get started today:

- Install the nightly build of TensorFlow.
- Check out the README (including known issues)
- Get detailed instructions in the eager execution User Guide
- Browse the eager examples in GitHub
- Follow the changelog for updates.

There's a lot more to talk about with eager execution and we're excited... or, rather, we're *eager* for you to try it today! Feedback is absolutely welcome.

27 条评论

Labels: Google Brain , Machine Learning , TensorFlow

TensorFlow Lattice: Flexibility Empowered by Prior Knowledge

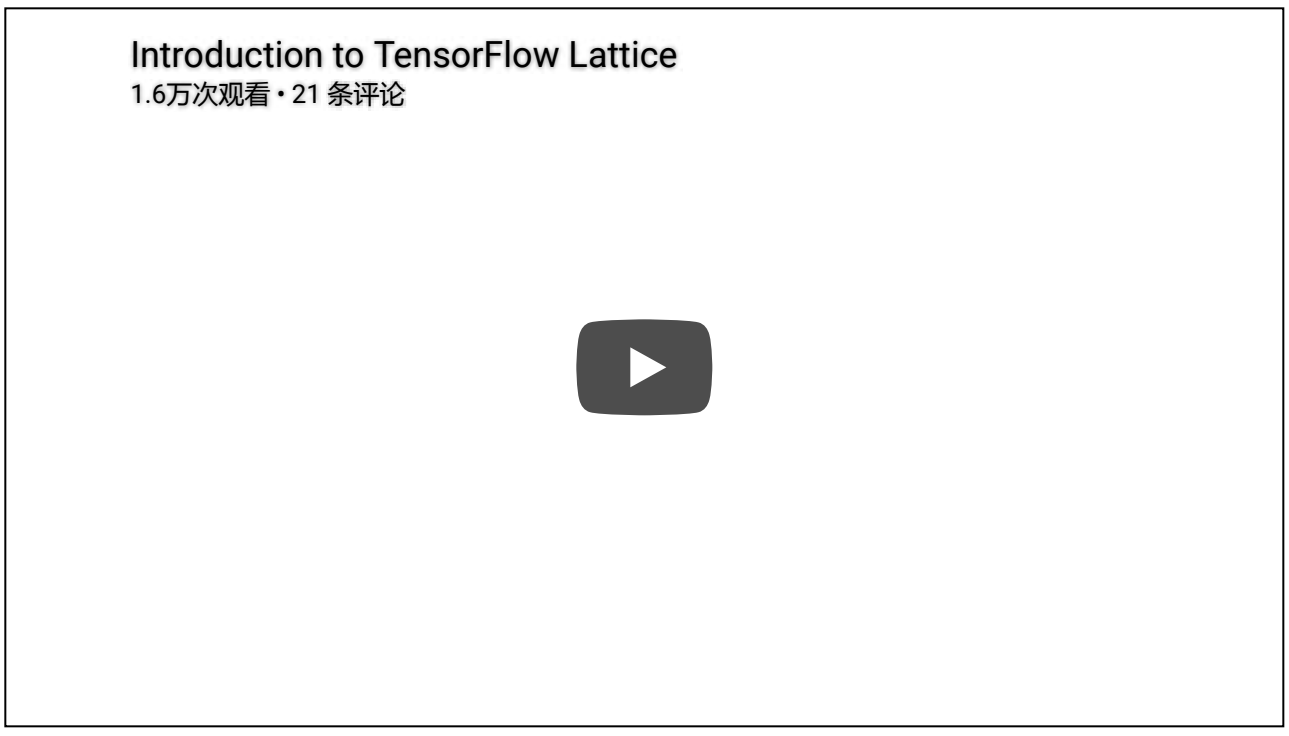
Wednesday, October 11, 2017

Posted by Maya Gupta, Research Scientist, Jan Pfeifer, Software Engineer and Seungil You, Software Engineer

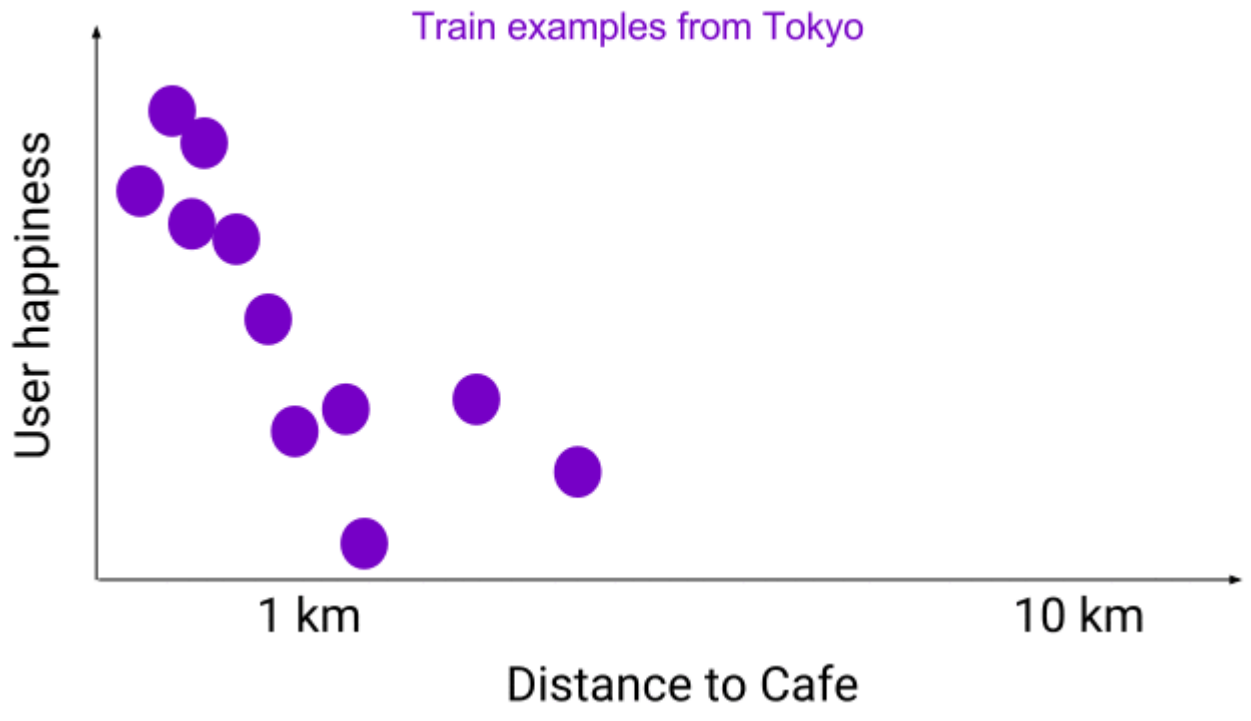
(Cross-posted on the Google Open Source Blog)

Machine learning has made huge advances in many applications including natural language processing, computer vision and recommendation systems by capturing complex input/output relationships using highly flexible models. However, a remaining challenge is problems with semantically meaningful inputs that obey known global relationships, like “the estimated time to drive a road goes up if traffic is heavier, and all else is the same.” Flexible models like DNNs and random forests may not learn these relationships, and then may fail to generalize well to examples drawn from a different sampling distribution than the examples the model was trained on.

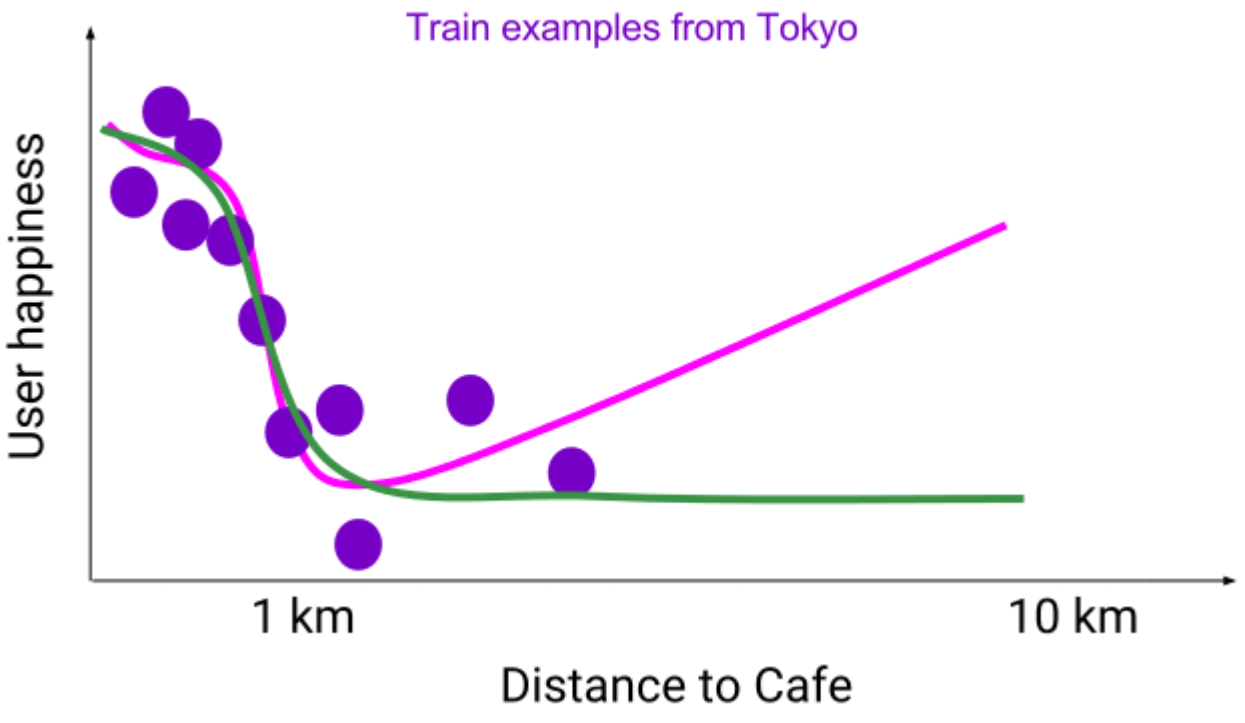
Today we present TensorFlow Lattice 格子, a set of prebuilt TensorFlow Estimators that are easy to use, and TensorFlow operators to build your own lattice models. Lattices are multi-dimensional interpolated look-up tables (for more details, see [1--5]), similar to the look-up tables in the back of a geometry textbook that approximate a sine function. We take advantage of the look-up table’s structure, which can be keyed by multiple inputs to approximate an arbitrarily flexible relationship, to satisfy monotonic relationships that you specify in order to generalize better. That is, the look-up table values are trained to minimize the loss on the training examples, but in addition, adjacent values in the look-up table are constrained to increase along given directions of the input space, which makes the model outputs increase in those directions. Importantly, because they interpolate between the look-up table values, the lattice models are smooth and the predictions are bounded, which helps to avoid spurious large or small predictions in the testing time.



How Lattice Models Help You
Suppose you are designing a system to recommend nearby coffee shops to a user. You would like the model to learn, “if two cafes are the same, prefer the closer one.” Below we show a flexible model (pink) that accurately fits some training data for users in Tokyo (purple), where there are many coffee shops nearby. The pink flexible model overfits the noisy training examples, and misses the overall trend that a closer cafe is better. If you used this pink model to rank test examples from Texas (blue), where businesses are spread farther out, you would find it acted strangely, sometimes preferring farther cafes!



Slice through a model’s feature space where all the other inputs stay the same and only distance changes. A flexible function (pink) that is accurate on training examples from Tokyo (purple) predicts that a cafe 10km-away is better than the same cafe if it was 5km-away. This problem becomes more evident at test-time if the data distribution has shifted, as shown here with blue examples from Texas where cafes are spread out more.



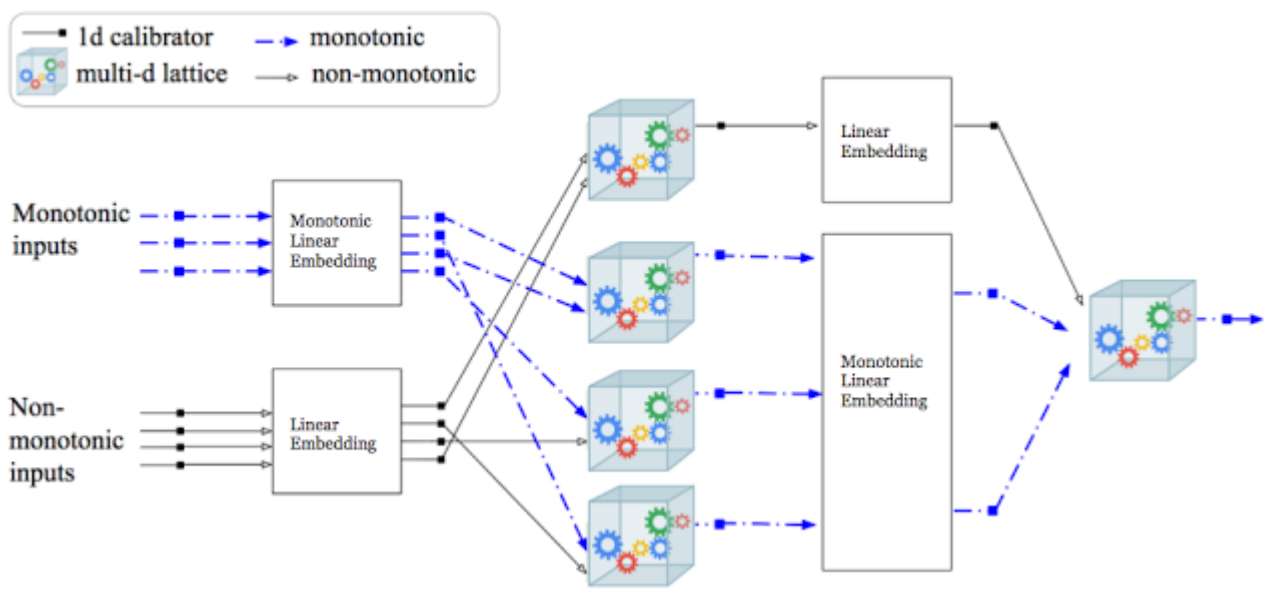
A monotonic flexible function (green) is both accurate on training examples and can generalize for Texas examples compared to non-monotonic flexible function (pink) from the previous figure.

In contrast, a lattice model, trained over the same example from Tokyo, can be constrained to satisfy such a monotonic relationship and result in a monotonic flexible function (green). The green line also accurately fits the Tokyo training examples, but also generalizes well to Texas, never preferring farther cafes.

In general, you might have many inputs about each cafe, e.g., coffee quality, price, etc. Flexible models have a hard time capturing global relationships of the form, “if all other inputs are equal, nearer is better, ” especially in parts of the feature space where your training data is sparse and noisy. Machine learning models that capture prior knowledge (e.g. how inputs should impact the prediction) work better in practice, and are easier to debug and more interpretable.

Pre-built Estimators
We provide a range of lattice model architectures as TensorFlow Estimators. The simplest estimator we provide is the *calibrated linear model*, which learns the best 1-d transformation of each feature (using 1-d lattices), and then combines all the calibrated features linearly. This works well if the training dataset is very small, or there are no complex nonlinear input interactions. Another estimator is a *calibrated lattice model*. This model combines the calibrated features nonlinearly using a two-layer single lattice model, which can represent complex nonlinear interactions in your dataset. The calibrated lattice model is usually a good choice if you have 2-10 features, but for 10 or more features, we expect you will get the best results with an ensemble of calibrated lattices, which you can train using the pre-built ensemble architectures. Monotonic lattice ensembles can achieve 0.3% -- 0.5% accuracy gain compared to Random Forests [4], and these new TensorFlow lattice estimators can achieve 0.1 -- 0.4% accuracy gain compared to the prior state-of-the-art in learning models with monotonicity [5].

Build Your Own
You may want to experiment with deeper lattice networks or research using partial monotonic functions as part of a deep neural network or other TensorFlow architecture. We provide the building blocks: TensorFlow operators for calibrators, lattice interpolation, and monotonicity projections. For example, the figure below shows a 9-layer deep lattice network [5].



Example of a 9-layer deep lattice network architecture [5], alternating layers of linear embeddings and ensembles of lattices with calibrators layers (which act like a sum of ReLU’s in Neural Networks). The blue lines correspond to monotonic inputs, which is preserved layer-by-layer, and hence for the entire model. This and other arbitrary architectures can be constructed with TensorFlow Lattice because each layer is differentiable.

In addition to the choice of model flexibility and standard L1 and L2 regularization, we offer new regularizers with TensorFlow Lattice:

- Monotonicity constraints [3] on your choice of inputs as described above.
- Laplacian regularization [3] on the lattices to make the learned function flatter.
- Torsion regularization [3] to suppress un-necessary nonlinear feature interactions.

We hope TensorFlow Lattice will be useful to the larger community working with meaningful semantic inputs. This is part of a larger research effort on interpretability and controlling machine learning models to satisfy policy goals, and enable practitioners to take advantage of their prior knowledge. We're excited to share this with all of you. To get started, please check out our GitHub repository and our tutorials, and let us know what you think!

Acknowledgements
Developing and open sourcing TensorFlow Lattice was a huge team effort. We'd like to thank all the people involved: Andrew Cotter, Kevin Canini, David Ding, Mahdi Milani Fard, Yifei Feng, Josh Gordon, Kiril Gorovoy, Clemens Mewald, Taman Narayan, Alexandre Passos, Christine Robson, Serena Wang, Martin Wicke, Jarek Wilkiewicz, Sen Zhao, Tao Zhu

References
[1] Lattice Regression, *Eric Garcia, Maya Gupta, Advances in Neural Information Processing Systems (NIPS), 2009*
[2] Optimized Regression for Efficient Function Evaluation, *Eric Garcia, Raman Arora, Maya R. Gupta, IEEE Transactions on Image Processing, 2012*
[3] Monotonic Calibrated Interpolated Look-Up Tables, *Maya Gupta, Andrew Cotter, Jan Pfeifer, Konstantin Voevodski, Kevin Canini, Alexander Mangylov, Wojciech Moczydlowski, Alexander van Esbroeck, Journal of Machine Learning Research (JMLR), 2016*
[4] Fast and Flexible Monotonic Functions with Ensembles of Lattices, *Mahdi Milani Fard, Kevin Canini, Andrew Cotter, Jan Pfeifer, Maya Gupta, Advances in Neural Information Processing Systems (NIPS), 2016*
[5] Deep Lattice Networks and Partial Monotonic Functions, *Seungil You, David Ding, Kevin Canini, Jan Pfeifer, Maya R. Gupta, Advances in Neural Information Processing Systems (NIPS), 2017*

Labels: Machine Learning , open source , TensorFlow