Google Research Blog

The latest news from Research at Google

TFGAN: A Lightweight Library for Generative Adversarial Networks

Tuesday, December 12, 2017

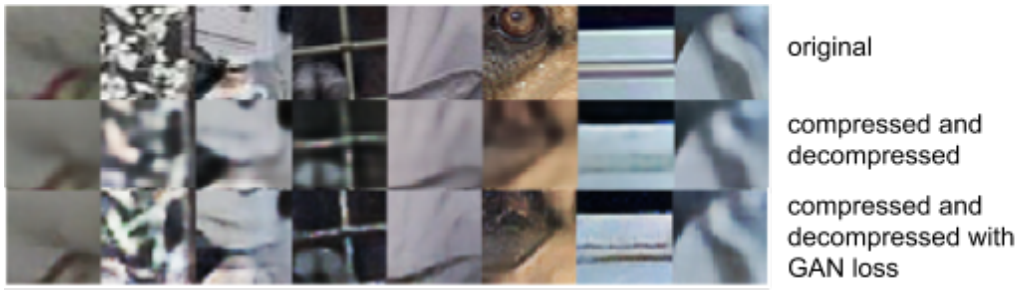
Posted by Joel Shor, Senior Software Engineer, Machine Perception

(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.

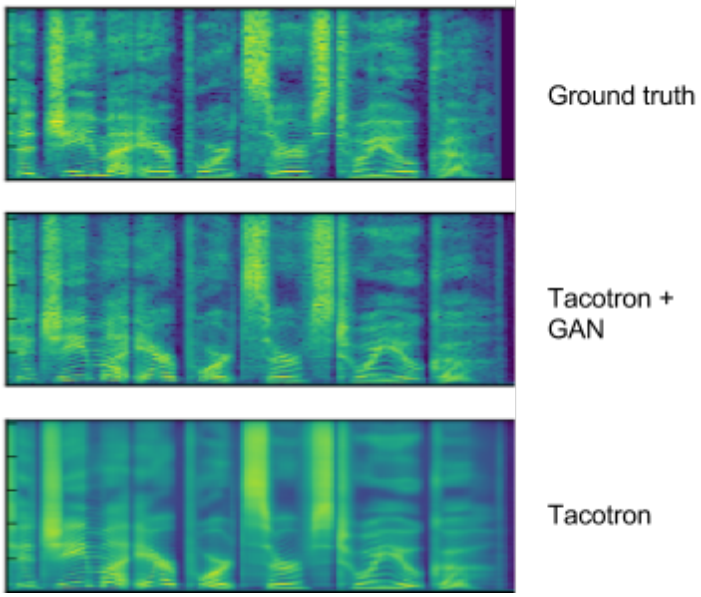
In order to make GANs easier to experiment with, we've open sourced TFGAN, a lightweight library designed to make it easy to train and evaluate GANs. It provides the infrastructure to easily train a GAN, provides well-tested loss and evaluation metrics, and gives easy-to-use examples that highlight the 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.

37 条

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

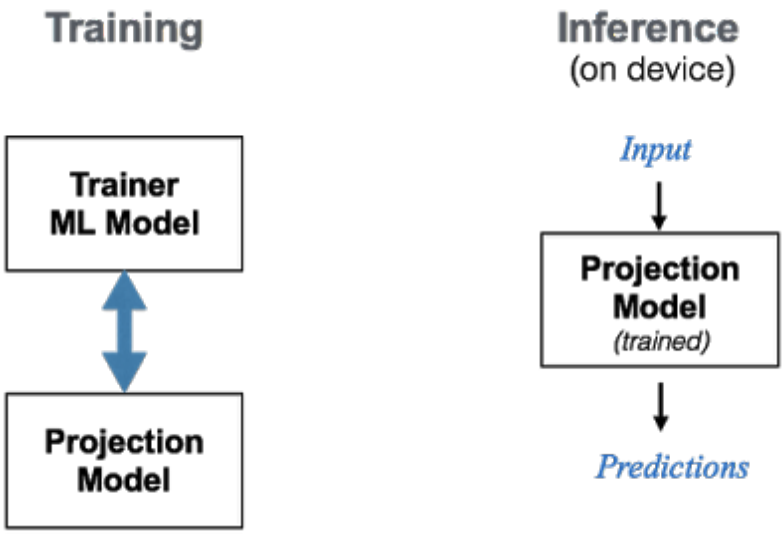
Earlier this year, we launched Android Wear 2.0 which featured the first "on-device" machine learning technology for smart messaging. This enabled cloud-based technologies like Smart Reply, previously available in Gmail, Inbox and Allo, to be used directly within any application for the first time, including 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.

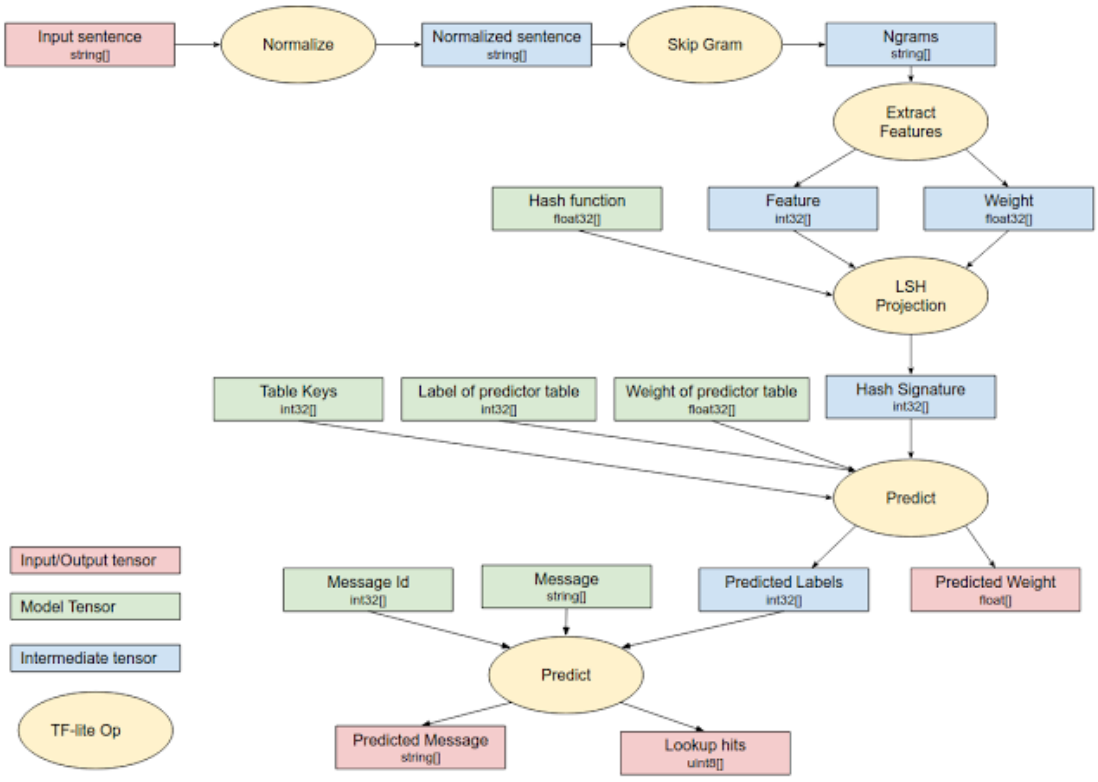
The on-device conversational model we have released uses a new ML architecture for training compact neural networks (as well as other machine learning models) based on a joint optimization framework, originally presented in ProjectionNet: Learning Efficient On-Device Deep Networks Using Neural Projections. 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.

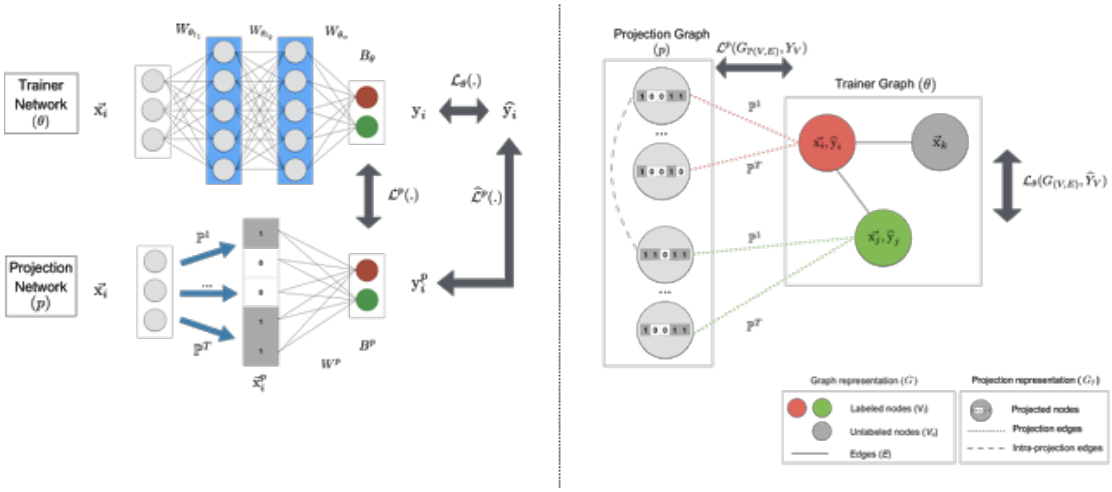
The open-source conversational model released today (along with code) was trained end-to-end using the joint ML architecture described above. Today's release also includes a demo app, so you can easily download and try out one-touch smart replies on your mobile device. The architecture enables easy configuration 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.

13 条

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.

26 条

Labels: Google Brain , Machine Learning , TensorFlow

TensorFlow Lattice: Flexibility Empowered by Prior Knowledge

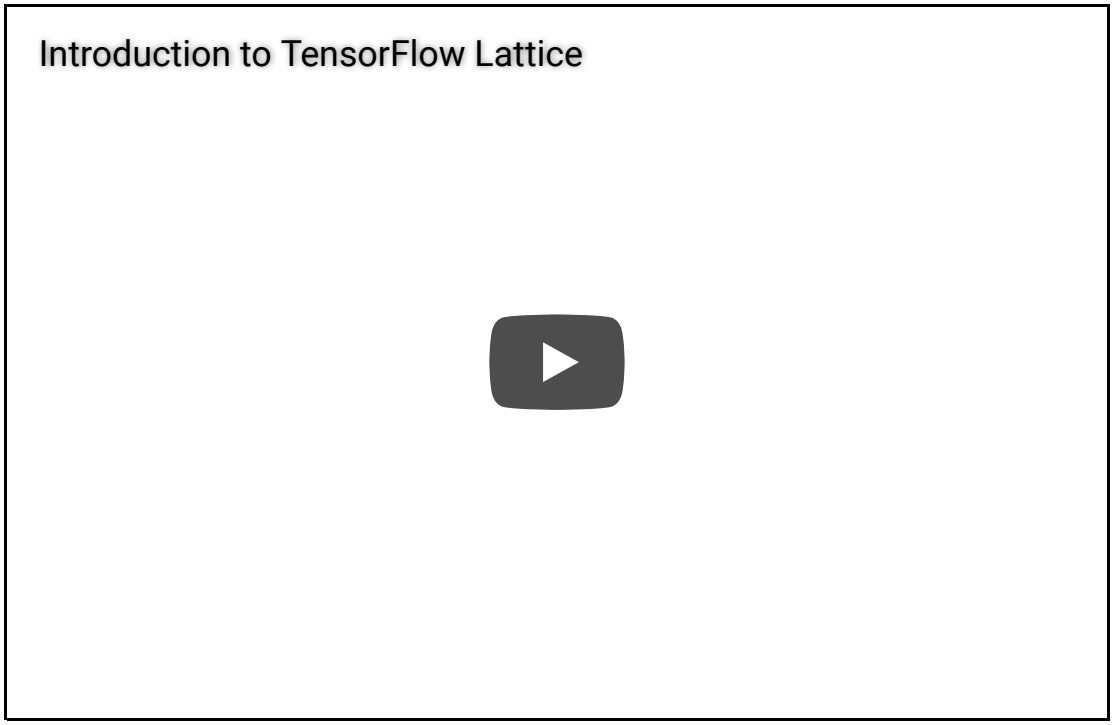
Wednesday, October 11, 2017

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

(Cross-posted on the Google Open Source Blog)

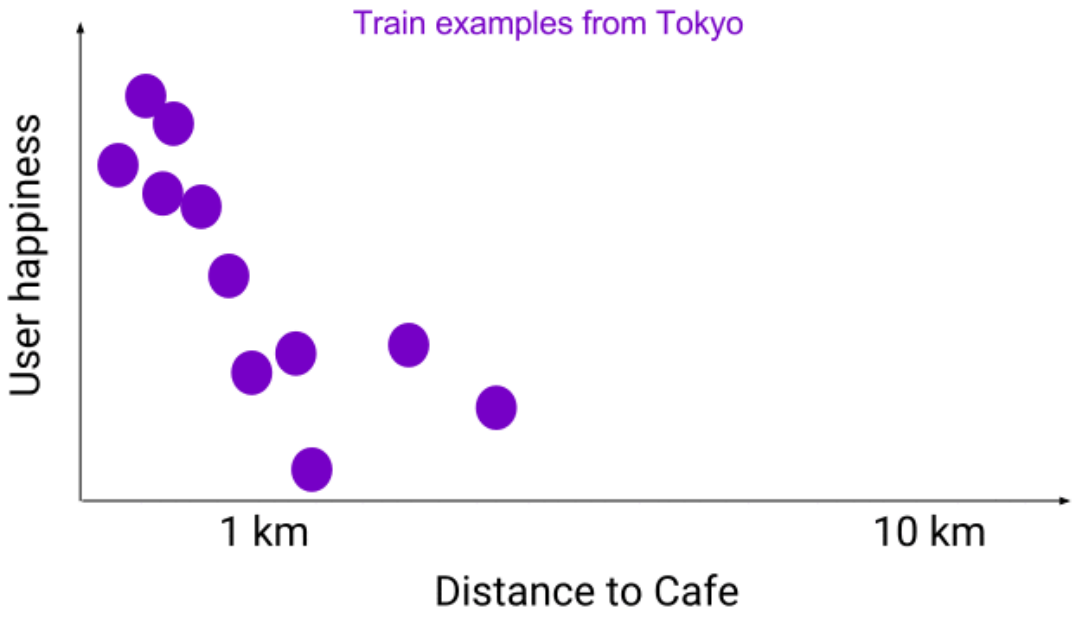
Machine learning has made huge advances in many applications including natural language processing, computer vision and recommendation systems by capturing complex input/output relationships using highly flexible models. However, a remaining challenge is problems with semantically meaningful inputs that 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, 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.

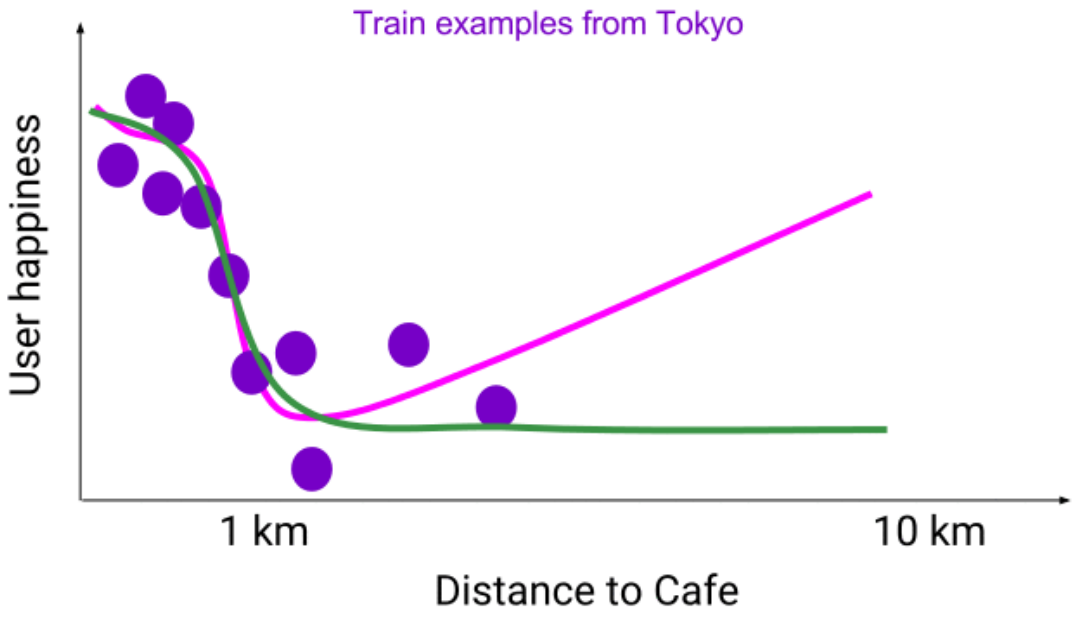


How Lattice Models Help You

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.

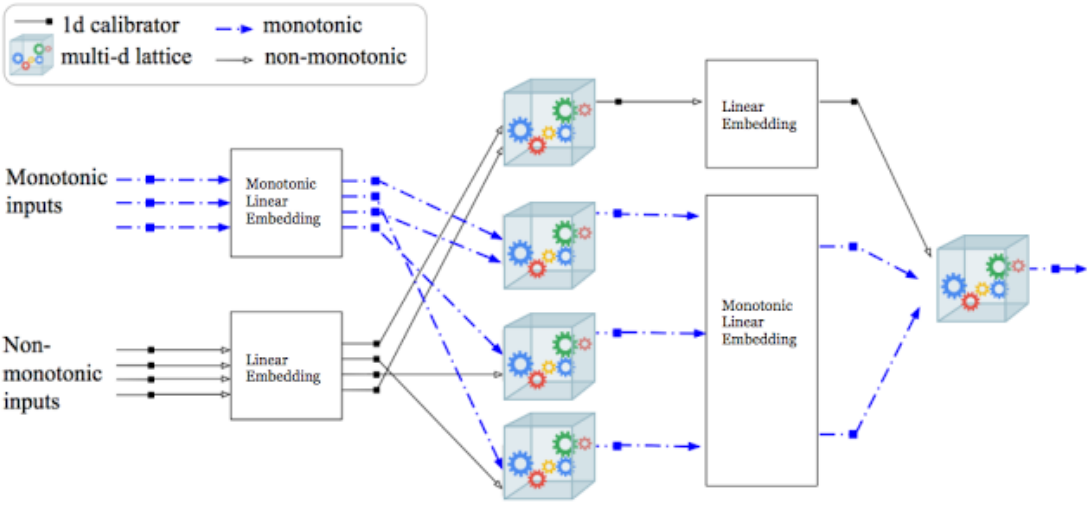
Pre-built Estimators

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

Build your own Machine Learning Visualizations

with the new TensorBoard API

Monday, September 11, 2017

Posted by Chi Zeng and Justine Tunney, Software Engineers, Google Brain Team

When we open-sourced TensorFlow in 2015, it included TensorBoard, a suite of visualizations for inspecting and understanding your TensorFlow models and runs. Tensorboard included a small, predetermined set of visualizations that are generic and applicable to nearly all deep learning applications such as observing how loss changes over time or exploring clusters in high-dimensional spaces. However, in the absence of reusable APIs, adding new visualizations to TensorBoard was prohibitively difficult for anyone outside of the TensorFlow team, leaving out a long tail of potentially creative, beautiful and useful visualizations that could be built by the research community.

To allow the creation of new and useful visualizations, we announce the release of a consistent set of APIs that allows developers to add custom visualization plugins to TensorBoard. We hope that developers use this API to extend TensorBoard and ensure that it covers a wider variety of use cases.

We have updated the existing dashboards (tabs) in TensorBoard to use the new API, so they serve as examples for plugin creators. For the current listing of plugins included within TensorBoard, you can explore the tensorboard/plugins directory on GitHub. For instance, observe the new plugin that generates precision-recall curves:

The plugin demonstrates the 3 parts of a standard TensorBoard plugin:

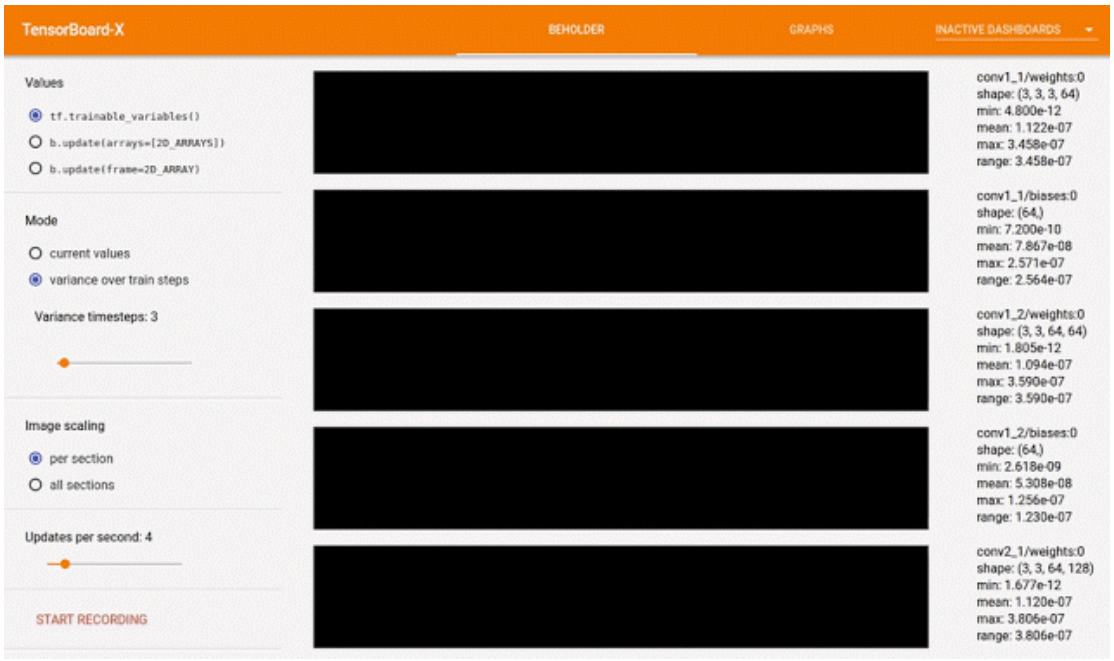
- A TensorFlow summary op used to collect data for later

visualization. [GitHub]

- A Python backend that serves custom data. [GitHub]
- A dashboard within TensorBoard built with TypeScript and polymer. [GitHub]

Additionally, like other plugins, the “pr_curves” plugin provides a demo that (1) users can look over in order to learn how to use the plugin and (2) the plugin author can use to generate example data during development. To further clarify how plugins work, we’ve also created a barebones TensorBoard “Greeter” plugin. This simple plugin collects greetings (simple strings preceded by “Hello, ”) during model runs and displays them. We recommend starting by exploring (or forking) the Greeter plugin as well as other existing plugins.

A notable example of how contributors are already using the TensorBoard API is Beholder, which was recently created by Chris Anderson while working on his master’s degree. Beholder shows a live video feed of data (e.g. gradients and convolution filters) as a model trains. You can watch the demo video here.



We look forward to seeing what innovations will come out of the community. If you plan to contribute a plugin to TensorBoard’s repository, you should get in touch with us first through the issue tracker with your idea so that we can help out and possibly guide you.

Acknowledgements

Dandelion Mané and William Chargin played crucial roles in building this API.

35 条

Labels: Google Brain , Machine Learning , open source , TensorBoard , TensorFlow , UI , Visualization

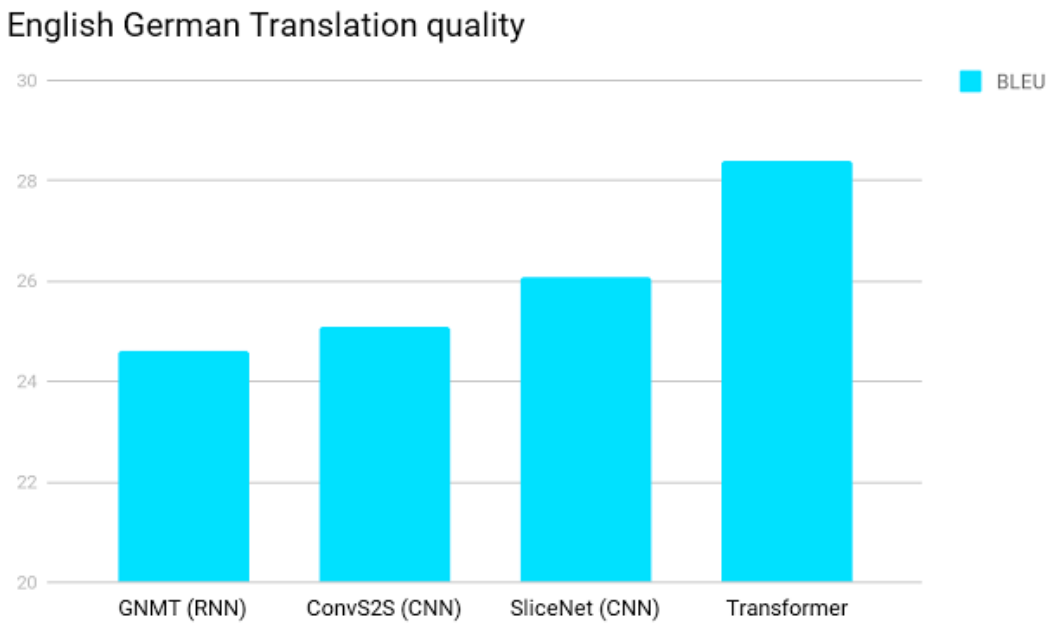
Transformer: A Novel Neural Network Architecture for Language Understanding

Thursday, August 31, 2017

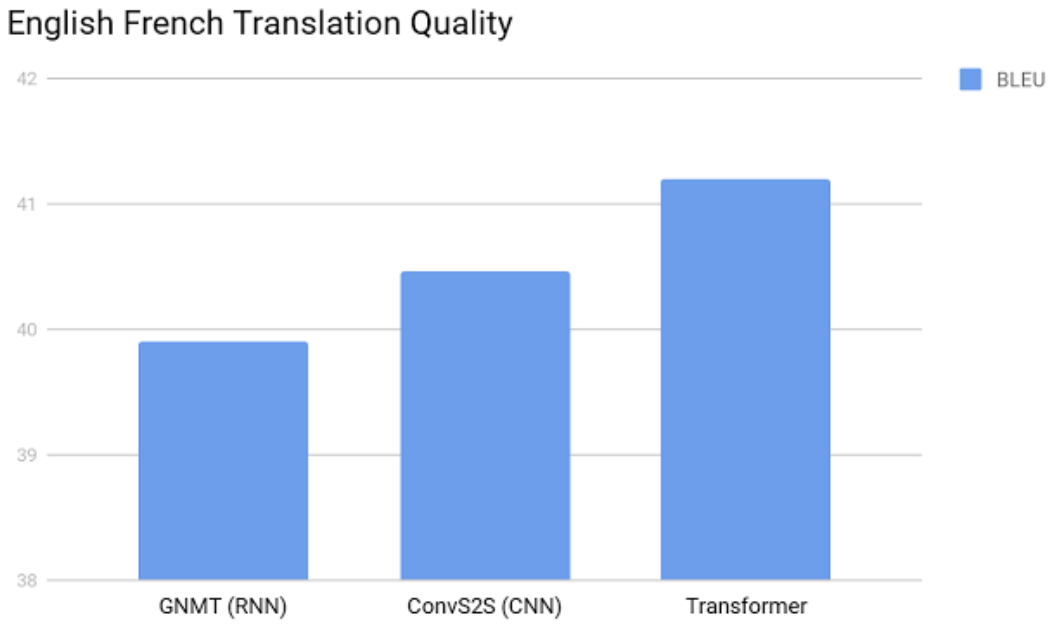
Posted by Jakob Uszkoreit, Software Engineer, Natural Language Understanding

Neural networks, in particular recurrent neural networks (RNNs), are now at the core of the leading approaches to language understanding tasks such as language modeling, machine translation and question answering. In Attention Is All You Need we introduce the Transformer, a novel neural network architecture based on a self-attention mechanism that we believe to be particularly well-suited for language understanding.

In our paper, we show that the Transformer outperforms both recurrent and convolutional models on academic English to German and English to French translation benchmarks. On top of higher translation quality, the Transformer requires less computation to train and is a much better fit for modern machine learning hardware, speeding up training by up to an order of magnitude.



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to German translation benchmark.



BLEU scores (higher is better) of single models on the standard WMT newstest2014 English to French translation benchmark.

Accuracy and Efficiency in Language Understanding

Neural networks usually process language by generating fixed- or variable-length vector-space representations. After starting with representations of individual words or even pieces of words, they aggregate information from surrounding words to determine the meaning of a given bit of language in context. For example, deciding on the most likely meaning and appropriate representation of the word “bank” in the sentence “I arrived at the bank after crossing the…” requires knowing if the sentence ends in “... road.” or “... river.”

RNNs have in recent years become the typical network architecture for translation, processing language sequentially in a left-to-right or right-to-left fashion. Reading one word at a time, this forces RNNs to perform multiple steps to make decisions that depend on words far away from each other. Processing the example above, an RNN could only determine that “bank” is likely to refer to the bank of a river after reading each word between “bank” and “river” step by step. Prior research has shown that, roughly speaking, the more such steps decisions require, the harder it is for a recurrent network to learn how to make those decisions.

The sequential nature of RNNs also makes it more difficult to fully take advantage of modern fast computing devices such as TPUs and GPUs, which excel at parallel and not sequential processing. Convolutional neural networks (CNNs) are much less sequential than RNNs, but in CNN architectures like ByteNet or ConvS2S the number of steps required to combine information from distant parts of the input still grows with increasing distance.

The Transformer

In contrast, the Transformer only performs a small, constant number of steps (chosen empirically). In each step, it applies a self-attention mechanism which directly models relationships between all words in a sentence, regardless of their respective position. In the earlier example “I arrived at the bank after crossing the river”, to determine that the word “bank” refers to the shore of a river and not a financial institution, the Transformer can learn to immediately attend to the word “river” and make this decision in a single step. In fact, in our English-French translation model we observe exactly this behavior.

More specifically, to compute the next representation for a given word - “bank” for example - the Transformer compares it to every other word in the sentence. The result of these comparisons is an attention score for every other word in the sentence. These attention scores determine how much each of the other words should contribute to the next representation of “bank”. In the example, the

disambiguating “river” could receive a high attention score when computing a new representation for “bank”. The attention scores are then used as weights for a weighted average of all words’ representations which is fed into a fully-connected network to generate a new representation for “bank”, reflecting that the sentence is talking about a river bank.

The animation below illustrates how we apply the Transformer to machine translation. Neural networks for machine translation typically contain an encoder reading the input sentence and generating a representation of it. A decoder then generates the output sentence word by word while consulting the representation generated by the encoder. The Transformer starts by generating initial representations, or embeddings, for each word. These are represented by the unfilled circles. Then, using self-attention, it aggregates information from all of the other words, generating a new representation per word informed by the entire context, represented by the filled balls. This step is then repeated multiple times in parallel for all words, successively generating new representations.

The decoder operates similarly, but generates one word at a time, from left to right.

It attends not only to the other previously generated words, but also to the final representations generated by the encoder.

Flow of Information

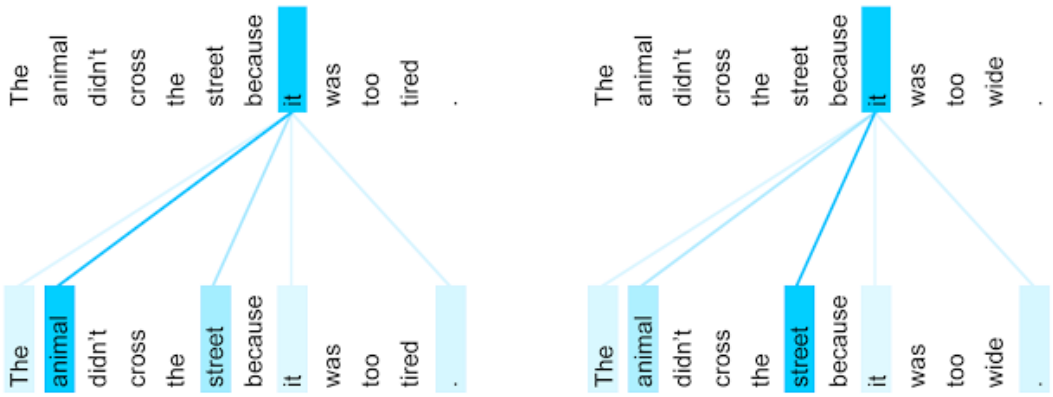
Beyond computational performance and higher accuracy, another intriguing aspect of the Transformer is that we can visualize what other parts of a sentence the network attends to when processing or translating a given word, thus gaining insights into how information travels through the network.

To illustrate this, we chose an example involving a phenomenon that is notoriously challenging for machine translation systems: coreference resolution. Consider the following sentences and their French translations:

The animal didn't cross the street because it was too tired.
L'animal n'a pas traversé la rue parce qu'il était trop fatigué.

The animal didn't cross the street because it was too wide.
L'animal n'a pas traversé la rue parce qu'elle était trop large.

It is obvious to most that in the first sentence pair “it” refers to the animal, and in the second to the street. When translating these sentences to French or German, the translation for “it” depends on the gender of the noun it refers to - and in French “animal” and “street” have different genders. In contrast to the current Google Translate model, the Transformer translates both of these sentences to French correctly. Visualizing what words the encoder attended to when computing the final representation for the word “it” sheds some light on how the network made the decision. In one of its steps, the Transformer clearly identified the two nouns “it” could refer to and the respective amount of attention reflects its choice in the different contexts.



The encoder self-attention distribution for the word “it” from the 5th to the 6th layer of a Transformer trained on English to French translation (one of eight attention heads).

Given this insight, it might not be that surprising that the Transformer also performs very well on the classic language analysis task of syntactic constituency parsing, a task the natural language processing community has attacked with highly specialized systems for decades.

In fact, with little adaptation, the same network we used for English to German translation outperformed all but one of the previously proposed approaches to constituency parsing.

Next Steps

We are very excited about the future potential of the Transformer and have already started applying it to other problems involving not only natural language but also very different inputs and outputs, such as images and video. Our ongoing experiments are accelerated immensely by the Tensor2Tensor library, which we recently open sourced. In fact, after downloading the library you can train your own Transformer networks for translation and parsing by invoking just a few commands. We hope you’ll give it a try, and look forward to seeing what the community can do with the Transformer.

Acknowledgements

This research was conducted by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser and Illia Polosukhin. Additional thanks go to David Chenell for creating the animation above.

Labels: Deep Learning , Machine Translation , Natural Language Processing , Natural Language Understanding , TensorFlow

Launching the Speech Commands Dataset

Thursday, August 24, 2017

Posted by Pete Warden, Software Engineer, Google Brain Team

At Google, we’re often asked how to get started using deep learning for speech and other audio recognition problems, like detecting keywords or commands. And while there are some great open source speech recognition systems like Kaldi that can

use neural networks as a component, their sophistication makes them tough to use as a guide to a simpler tasks. Perhaps more importantly, there aren't many free and openly available datasets ready to be used for a beginner's tutorial (many require preprocessing before a neural network model can be built on them) or that are well suited for simple keyword detection.

To solve these problems, the TensorFlow and AIY teams have created the Speech Commands Dataset, and used it to add training* and inference sample code to TensorFlow. The dataset has 65,000 one-second long utterances of 30 short words, by thousands of different people, contributed by members of the public through the AIY website. It's released under a Creative Commons BY 4.0 license, and will continue to grow in future releases as more contributions are received. The dataset is designed to let you build basic but useful voice interfaces for applications, with common words like "Yes", "No", digits, and directions included. The infrastructure we used to create the data has been open sourced too, and we hope to see it used by the wider community to create their own versions, especially to cover underserved languages and applications.

To try it out for yourself, download the prebuilt set of the TensorFlow Android demo applications and open up "TF Speech". You'll be asked for permission to access your microphone, and then see a list of ten words, each of which should light up as you say them.

The results will depend on whether your speech patterns are covered by the dataset, so it may not be perfect — commercial speech recognition systems are a lot more complex than this teaching example. But we're hoping that as more accents and variations are added to the dataset, and as the community contributes improved models to TensorFlow, we'll continue to see improvements and extensions.

You can also learn how to train your own version of this model through the new audio recognition tutorial on TensorFlow.org. With the latest development version of the framework and a modern desktop machine, you can download the dataset and train the model in just a few hours. You'll also see a wide variety of options to customize the neural network for different problems, and to make different latency, size, and accuracy tradeoffs to run on different platforms.

We are excited to see what new applications people are able to build with the help of this dataset and tutorial, so I hope you get a chance to dive in and start recognizing!

* The architecture this network is based on is described in Convolutional Neural Networks for Small-footprint Keyword Spotting, presented at Interspeech 2015. ↩

Labels: datasets , Google Brain , Speech Recognition , TensorFlow

Building Your Own Neural Machine Translation System in TensorFlow

Wednesday, July 12, 2017

Posted by Thang Luong, Research Scientist, and Eugene Brevdo, Staff Software Engineer, Google Brain Team

Machine translation – the task of automatically translating between languages – is one of the most active research areas in the machine learning community. Among the many approaches to machine translation, sequence-to-sequence ("seq2seq") models [1, 2] have recently enjoyed great success and have become the de facto standard in most commercial translation systems, such as Google Translate, thanks to its ability to use deep neural networks to capture sentence meanings. However, while there is an abundance of material on seq2seq models such as OpenNMT or tf-seq2seq, there is a lack of material that teaches people both the

knowledge and the skills to easily build high-quality translation systems.

Today we are happy to announce a new Neural Machine Translation (NMT) tutorial for TensorFlow that gives readers a full understanding of seq2seq models and shows how to build a competitive translation model from scratch. The tutorial is aimed at making the process as simple as possible, starting with some background knowledge on NMT and walking through code details to build a vanilla system. It then dives into the attention mechanism [3, 4], a key ingredient that allows NMT systems to handle long sentences. Finally, the tutorial provides details on how to replicate key features in the Google's NMT (GNMT) system [5] to train on multiple GPUs.

The tutorial also contains detailed benchmark results, which users can replicate on their own. Our models provide a strong open-source baseline with performance on par with GNMT results [5]. We achieve 24.4 BLEU points on the popular WMT'14 English-German translation task.

Other benchmark results (English-Vietnamese, German-English) can be found in the tutorial.

In addition, this tutorial showcases the fully dynamic seq2seq API (released with TensorFlow 1.2) aimed at making building seq2seq models clean and easy:

- Easily read and preprocess dynamically sized input sequences using the new input pipeline in tf.contrib.data.
- Use padded batching and sequence length bucketing to improve training and inference speeds.
- Train seq2seq models using popular architectures and training schedules, including several types of attention and scheduled sampling.
- Perform inference in seq2seq models using in-graph beam search.
- Optimize seq2seq models for multi-GPU settings.

We hope this will help spur the creation of, and experimentation with, many new NMT models by the research community. To get started on your own research, check out the tutorial on GitHub!

Core contributors

Thang Luong, Eugene Brevdo, and Rui Zhao.

Acknowledgements

We would like to especially thank our collaborator on the NMT project, Rui Zhao. Without his tireless effort, this tutorial would not have been possible. Additional

thanks go to Denny Britz, Anna Goldie, Derek Murray, and Cinjon Resnick for their work bringing new features to TensorFlow and the seq2seq library. Lastly, we thank Lukasz Kaiser for the initial help on the seq2seq codebase; Quoc Le for the suggestion to replicate GNMT; Yonghui Wu and Zhifeng Chen for details on the GNMT systems; as well as the Google Brain team for their support and feedback!

References

[1] Sequence to sequence learning with neural networks, *Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. NIPS, 2014.*

[2] Learning phrase representations using RNN encoder-decoder for statistical machine translation, *Kyunghyun Cho, Bart Van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. EMNLP 2014.*

[3] Neural machine translation by jointly learning to align and translate, *Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. ICLR, 2015.*

[4] Effective approaches to attention-based neural machine translation, *Minh-Thang Luong, Hieu Pham, and Christopher D Manning. EMNLP, 2015.*

[5] Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation, *Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Łukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, Jeffrey Dean. Technical Report, 2016.*

Labels: Google Brain , Machine Translation , Neural Networks , open source , TensorFlow

MultiModel: Multi-Task Machine Learning Across Domains

Wednesday, June 21, 2017

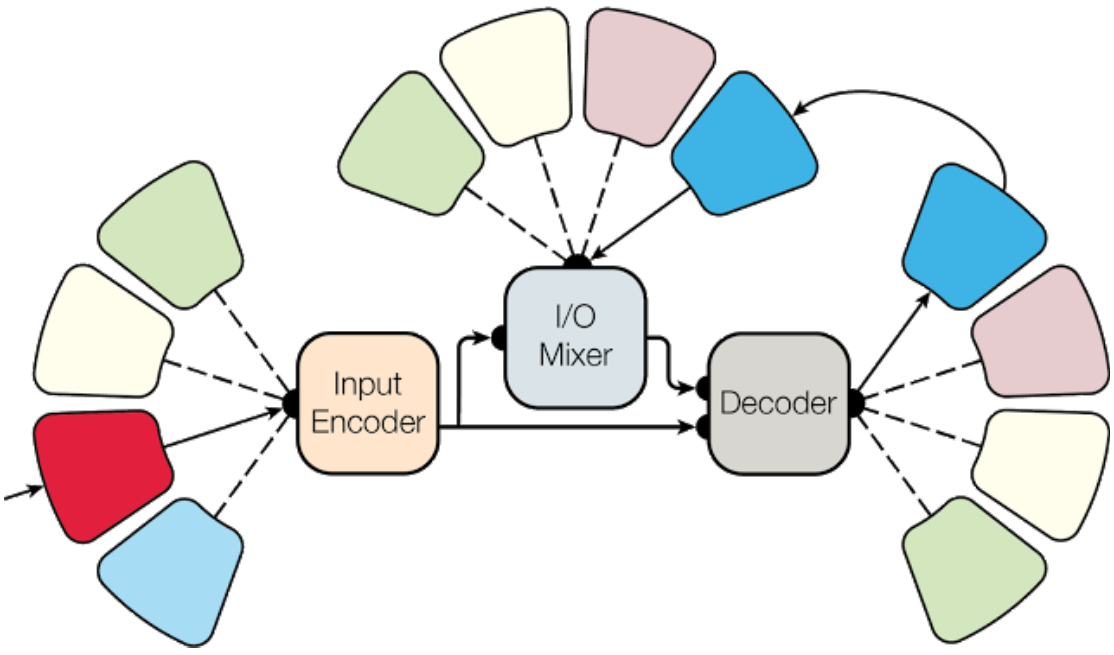
Posted by Łukasz Kaiser, Senior Research Scientist, Google Brain Team and Aidan N. Gomez, Researcher, Department of Computer Science Machine Learning Group, University of Toronto

Over the last decade, the application and performance of Deep Learning has progressed at an astonishing rate. However, the current state of the field is that the

neural network architectures are highly specialized to specific domains of application. An important question remains unanswered: Will a convergence between these domains facilitate a unified model capable of performing well across multiple domains?

Today, we present MultiModel, a neural network architecture that draws from the success of vision, language and audio networks to simultaneously solve a number of problems spanning multiple domains, including image recognition, translation and speech recognition. While strides have been made in this direction before, namely in Google's Multilingual Neural Machine Translation System used in Google Translate, MultiModel is a first step towards the convergence of vision, audio and language understanding into a single network.

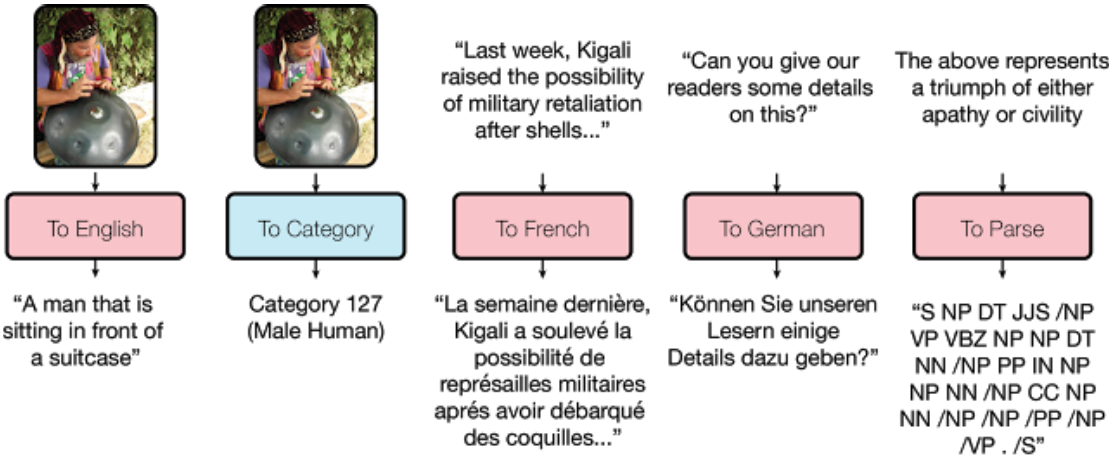
The inspiration for how MultiModel handles multiple domains comes from how the brain transforms sensory input from different modalities (such as sound, vision or taste), into a single shared representation and back out in the form of language or actions. As an analog to these modalities and the transformations they perform, MultiModel has a number of small modality-specific sub-networks for audio, images, or text, and a shared model consisting of an encoder, input/output mixer and decoder, as illustrated below.



MultiModel architecture: small modality-specific sub-networks work with a shared encoder, I/O mixer and decoder. Each petal represents a modality, transforming to and from the internal representation.

We demonstrate that MultiModel is capable of learning eight different tasks

simultaneously: it can detect objects in images, provide captions, recognize speech, translate between four pairs of languages, and do grammatical constituency parsing at the same time. The input is given to the model together with a very simple signal that determines which output we are requesting. Below we illustrate a few examples taken from a MultiModel trained jointly on these eight tasks[1]:



When designing MultiModel it became clear that certain elements from each domain of research (vision, language and audio) were integral to the model's success in related tasks. We demonstrate that these computational primitives (such as convolutions, attention, or mixture-of-experts layers) clearly improve performance on their originally intended domain of application, while not hindering MultiModel's performance on other tasks. It is not only possible to achieve good performance while training jointly on multiple tasks, but on tasks with limited quantities of data, the performance actually improves. To our surprise, this happens even if the tasks come from different domains that would appear to have little in common, e.g., an image recognition task can improve performance on a language task.

It is important to note that while MultiModel does not establish new performance records, it does provide insight into the dynamics of multi-domain multi-task learning in neural networks, and the potential for improved learning on data-limited tasks by the introduction of auxiliary tasks. There is a longstanding saying in machine learning: "the best regularizer is more data"; in MultiModel, this data can be sourced across domains, and consequently can be obtained more easily than previously thought. MultiModel provides evidence that training in concert with other tasks can lead to good results and improve performance on data-limited tasks.

Many questions about multi-domain machine learning remain to be studied, and we will continue to work on tuning Multimodel and improving its performance. To allow this research to progress quickly, we open-sourced MultiModel as part of the Tensor2Tensor library. We believe that such synergetic models trained on data from

multiple domains will be the next step in deep learning and will ultimately solve tasks beyond the reach of current narrowly trained networks.

Acknowledgements

This work is a collaboration between Googlers Łukasz Kaiser, Noam Shazeer, Ashish Vaswani, Niki Parmar, Llion Jones and Jakob Uszkoreit, and Aidan N. Gomez from the University of Toronto. It was performed while Aidan was working with the Google Brain team.

---

1 The 8 tasks were: (1) speech recognition (WSJ corpus), (2) image classification (ImageNet), (3) image captioning (MS COCO), (4) parsing (Penn Treebank), (5) English-German translation, (6) German-English translation, (7) English-French translation, (8) French-English translation (all using WMT data-sets). ↩

116 条评



Labels: Deep Learning , Google Brain , Natural Language Processing , open source , TensorFlow

Accelerating Deep Learning Research with the Tensor2Tensor Library

Monday, June 19, 2017

Posted by Łukasz Kaiser, Senior Research Scientist, Google Brain Team

Deep Learning (DL) has enabled the rapid advancement of many useful technologies, such as machine translation, speech recognition and object detection. In the research community, one can find code open-sourced by the authors to help in replicating their results and further advancing deep learning. However, most of these DL systems use unique setups that require significant engineering effort and may only work for a specific problem or architecture, making it hard to run new experiments and compare the results.

Today, we are happy to release Tensor2Tensor (T2T), an open-source system for training deep learning models in TensorFlow. T2T facilitates the creation of state-of-the art models for a wide variety of ML applications, such as translation, parsing, image captioning and more, enabling the exploration of various ideas much faster than previously possible. This release also includes a library of datasets and models, including the best models from a few recent papers (*Attention Is All You Need*, *Depthwise Separable Convolutions for Neural Machine Translation* and *One*

Model to Learn Them All) to help kick-start your own DL research.

Translation Model	Training time	BLEU (difference from baseline)
Transformer (T2T)	3 days on 8 GPU	**28.4 (+7.8)**
SliceNet (T2T)	6 days on 32 GPUs	26.1 (+5.5)
GNMT + Mixture of Experts	1 day on 64 GPUs	26.0 (+5.4)
ConvS2S	18 days on 1 GPU	25.1 (+4.5)
GNMT	1 day on 96 GPUs	24.6 (+4.0)
ByteNet	8 days on 32 GPUs	23.8 (+3.2)
MOSES (phrase-based baseline)	N/A	20.6 (+0.0)

BLEU scores (higher is better) on the standard WMT English-German translation task.

As an example of the kind of improvements T2T can offer, we applied the library to machine translation. As you can see in the table above, two different T2T models, SliceNet and Transformer, outperform the previous state-of-the-art, GNMT+MoE. Our best T2T model, Transformer, is 3.8 points better than the standard GNMT model, which itself was 4 points above the baseline phrase-based translation system, MOSES. Notably, with T2T you can approach previous state-of-the-art results with a single GPU in one day: a small Transformer model (not shown above) gets 24.9 BLEU after 1 day of training on a single GPU. Now everyone with a GPU can tinker with great translation models on their own: our github repo has instructions on how to do that.

Modular Multi-Task Training

The T2T library is built with familiar TensorFlow tools and defines multiple pieces needed in a deep learning system: data-sets, model architectures, optimizers, learning rate decay schemes, hyperparameters, and so on. Crucially, it enforces a standard interface between all these parts and implements current ML best practices. So you can pick any data-set, model, optimizer and set of hyperparameters, and run the training to check how it performs. We made the architecture modular, so every piece between the input data and the predicted output is a tensor-to-tensor function. If you have a new idea for the model architecture, you don't need to replace the whole setup. You can keep the embedding part and the loss and everything else, just replace the model body by your own function that takes a tensor as input and returns a tensor.

This means that T2T is flexible, with training no longer pinned to a specific model or

dataset. It is so easy that even architectures like the famous LSTM sequence-to-sequence model can be defined in a few dozen lines of code. One can also train a single model on multiple tasks from different domains. Taken to the limit, you can even train a single model on all data-sets concurrently, and we are happy to report that our MultiModel, trained like this and included in T2T, yields good results on many tasks even when training jointly on ImageNet (image classification), MS COCO (image captioning), WSJ (speech recognition), WMT (translation) and the Penn Treebank parsing corpus. It is the first time a single model has been demonstrated to be able to perform all these tasks at once.

Built-in Best Practices

With this initial release, we also provide scripts to generate a number of data-sets widely used in the research community[1], a handful of models[2], a number of hyperparameter configurations, and a well-performing implementation of other important tricks of the trade. While it is hard to list them all, if you decide to run your model with T2T you'll get for free the correct padding of sequences and the corresponding cross-entropy loss, well-tuned parameters for the Adam optimizer, adaptive batching, synchronous distributed training, well-tuned data augmentation for images, label smoothing, and a number of hyper-parameter configurations that worked very well for us, including the ones mentioned above that achieve the state-of-the-art results on translation and may help you get good results too.

As an example, consider the task of parsing English sentences into their grammatical constituency trees. This problem has been studied for decades and competitive methods were developed with a lot of effort. It can be presented as a sequence-to-sequence problem and be solved with neural networks, but it used to require a lot of tuning. With T2T, it took us only a few days to add the parsing data-set generator and adjust our attention transformer model to train on this problem. To our pleasant surprise, we got very good results in only a week:

Parsing Model	F1 score (higher is better)
Transformer (T2T)	91.3
Dyer et al.	**91.7**
Zhu et al.	90.4
Socher et al.	90.4
Vinyals & Kaiser et al.	88.3

Parsing F1 scores on the standard test set, section 23 of the WSJ. We only compare here models trained discriminatively on the Penn Treebank WSJ training set, see the paper for more results.

Contribute to Tensor2Tensor

In addition to exploring existing models and data-sets, you can easily define your own model and add your own data-sets to Tensor2Tensor. We believe the already included models will perform very well for many NLP tasks, so just adding your data-set might lead to interesting results. By making T2T modular, we also make it very easy to contribute your own model and see how it performs on various tasks. In this way the whole community can benefit from a library of baselines and deep learning research can accelerate. So head to our github repository, try the new models, and contribute your own!

Acknowledgements

The release of Tensor2Tensor was only possible thanks to the widespread collaboration of many engineers and researchers. We want to acknowledge here the core team who contributed (in alphabetical order): *Samy Bengio, Eugene Brevdo, Francois Chollet, Aidan N. Gomez, Stephan Gouws, Llion Jones, Łukasz Kaiser, Nal Kalchbrenner, Niki Parmar, Ryan Sepassi, Noam Shazeer, Jakob Uszkoreit, Ashish Vaswani*.

---

1 We include a number of datasets for image classification (MNIST, CIFAR-10, CIFAR-100, ImageNet), image captioning (MS COCO), translation (WMT with multiple languages including English-German and English-French), language modelling (LM1B), parsing (Penn Treebank), natural language inference (SNLI), speech recognition (TIMIT), algorithmic problems (over a dozen tasks from reversing through addition and multiplication to algebra) and we will be adding more and welcome your data-sets too.↩

2 Including LSTM sequence-to-sequence RNNs, convolutional networks also with separable convolutions (e.g., Xception), recently researched models like ByteNet or the Neural GPU, and our new state-of-the-art models mentioned in this post that we will be actively updating in the repository.↩

Labels: Deep Learning , Google Brain , Natural Language Processing , open source , TensorFlow , Translate

Supercharge your Computer Vision models with the TensorFlow Object Detection API

Thursday, June 15, 2017

Posted by Jonathan Huang, Research Scientist and Vivek Rathod, Software Engineer

(Cross-posted on the Google Open Source Blog)

At Google, we develop flexible state-of-the-art machine learning (ML) systems for computer vision that not only can be used to improve our products and services, but also spur progress in the research community. Creating accurate ML models capable of localizing and identifying multiple objects in a single image remains a core challenge in the field, and we invest a significant amount of time training and experimenting with these systems.

Detected objects in a sample image (from the COCO dataset) made by one of our models. Image credit: Michael Miley, original image.

Last October, our in-house object detection system achieved new state-of-the-art results, and placed first in the COCO detection challenge. Since then, this system has generated results for a number of research publications[1,2,3,4,5,6,7] and has been put to work in Google products such as NestCam, the similar items and style ideas feature in Image Search and street number and name detection in Street View.

Today we are happy to make this system available to the broader research community via the TensorFlow Object Detection API. This codebase is an open-source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. Our goals in designing this system was to

support state-of-the-art models while allowing for rapid exploration and research. Our first release contains the following:

- A selection of trainable detection models, including:
  - Single Shot Multibox Detector (SSD) with MobileNets
  - SSD with Inception V2
  - Region-Based Fully Convolutional Networks (R-FCN) with Resnet 101
  - Faster RCNN with Resnet 101
  - Faster RCNN with Inception Resnet v2
- Frozen weights (trained on the COCO dataset) for each of the above models to be used for out-of-the-box inference purposes.
- A Jupyter notebook for performing out-of-the-box inference with one of our released models
- Convenient local training scripts as well as distributed training and evaluation pipelines via Google Cloud

The SSD models that use MobileNet are lightweight, so that they can be comfortably run in real time on mobile devices. Our winning COCO submission in 2016 used an ensemble of the Faster RCNN models, which are more computationally intensive but significantly more accurate. For more details on the performance of these models, see our CVPR 2017 paper.

Are you ready to get started?

We've certainly found this code to be useful for our computer vision needs, and we hope that you will as well. Contributions to the codebase are welcome and please stay tuned for our own further updates to the framework. To get started, download the code here and try detecting objects in some of your own images using the Jupyter notebook, or training your own pet detector on Cloud ML engine!

Acknowledgements

The release of the Tensorflow Object Detection API and the pre-trained model zoo has been the result of widespread collaboration among Google researchers with feedback and testing from product groups. In particular we want to highlight the contributions of the following individuals:

Core Contributors: *Derek Chow, Chen Sun, Menglong Zhu, Matthew Tang, Anoop Korattikara, Alireza Fathi, Ian Fischer, Zbigniew Wojna, Yang Song, Sergio Guadarrama, Jasper Uijlings, Viacheslav Kovalevskyi, Kevin Murphy*

Also special thanks to: *Andrew Howard, Rahul Sukthankar, Vittorio Ferrari, Tom Duerig, Chuck Rosenberg, Hartwig Adam, Jing Jing Long, Victor Gomes, George Papandreou, Tyler Zhu*

References

1. Speed/accuracy trade-offs for modern convolutional object detectors, *Huang et al., CVPR 2017 (paper describing this framework)*
2. Towards Accurate Multi-person Pose Estimation in the Wild, *Papandreou et al., CVPR 2017*
3. YouTube-BoundingBoxes: A Large High-Precision Human-Annotated Data Set for Object Detection in Video, *Real et al., CVPR 2017 (see also our blog post)*
4. Beyond Skip Connections: Top-Down Modulation for Object Detection, *Shrivastava et al., arXiv preprint arXiv:1612.06851, 2016*
5. Spatially Adaptive Computation Time for Residual Networks, *Figurnov et al., CVPR 2017*
6. AVA: A Video Dataset of Spatio-temporally Localized Atomic Visual Actions, *Gu et al., arXiv preprint arXiv:1705.08421, 2017*
7. MobileNets: Efficient convolutional neural networks for mobile vision applications, *Howard et al., arXiv preprint arXiv:1704.04861, 2017*

Labels: Computer Vision , Deep Learning , Machine Learning , Machine Perception , open source , TensorFlow