Google AI
Brain Team

Reliable Deep Learning with Edward2 & Uncertainty Baselines

Dustin Tran Zachary Nado

Google AI
Brain Team

The Reliable Deep Learning Team at Google


balajiln


trandustin


jsnoek


dsculley


kpmurphy


adlam


jjren


kehanghan


dusenberrymw


rjenatton


shreyaspadhy


znado


zmariet


wangzi


knix


smalling


jereliu


markcollier


agroy


fortuin


jallingham


nehagup


phandu


ghassen

+many wonderful collaborators (Google Research Zurich, Oxford, Assistant, Health, ...)

Our mission

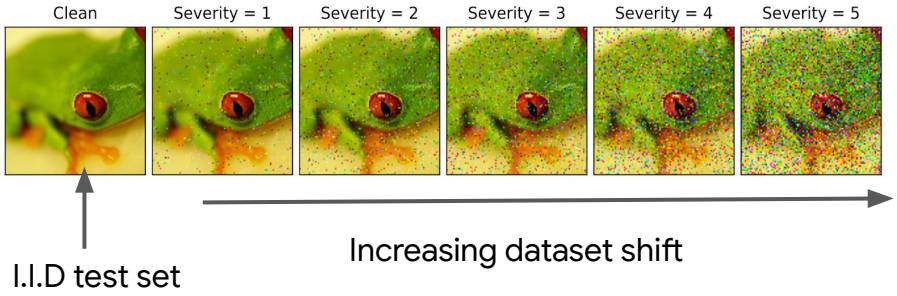
Develop models that ***“know what they don’t know”***.

This requires

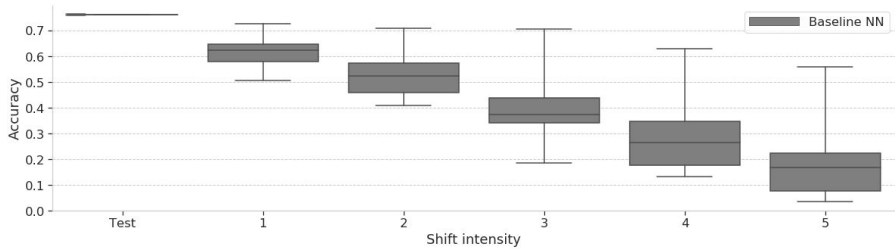
robustness to **distribution shift** and

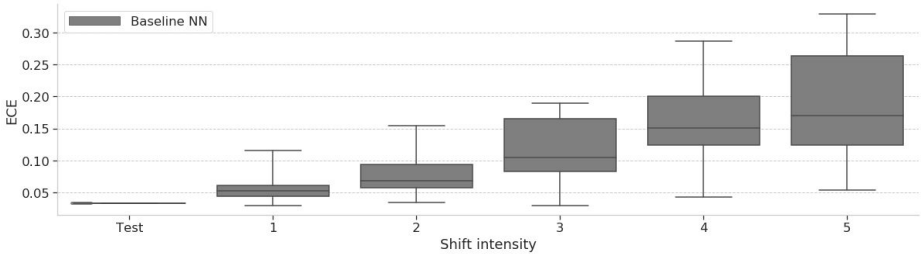
calibrated uncertainty quantification.

Why Reliable Deep Learning?



Out-of-distribution (OOD): $p_{TEST}(y,x) \neq p_{TRAIN}(y,x)$





- Accuracy of NNs degrades under dataset shift.
- Calibration also degrades under dataset shift.

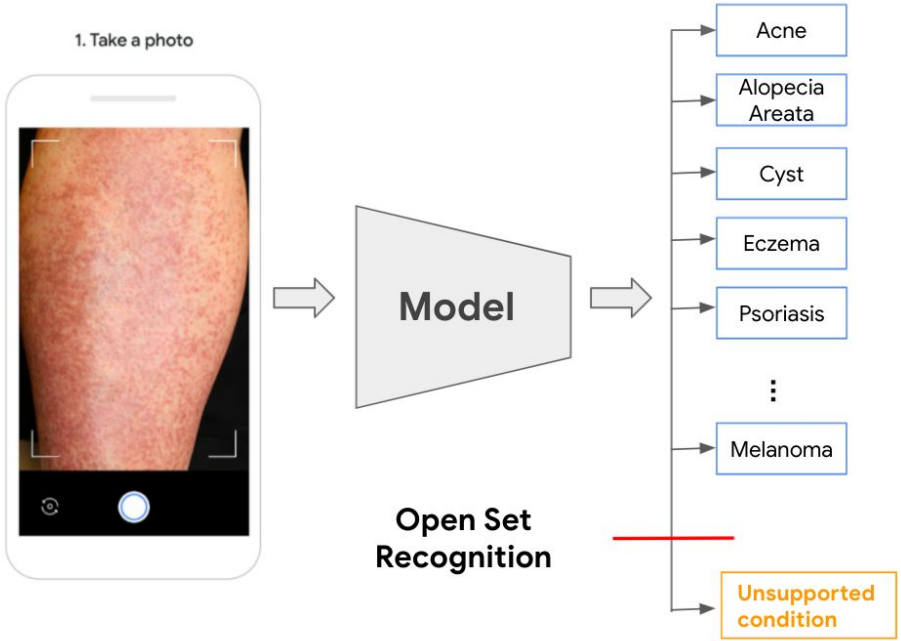
Calibration Error = |Confidence - Accuracy|

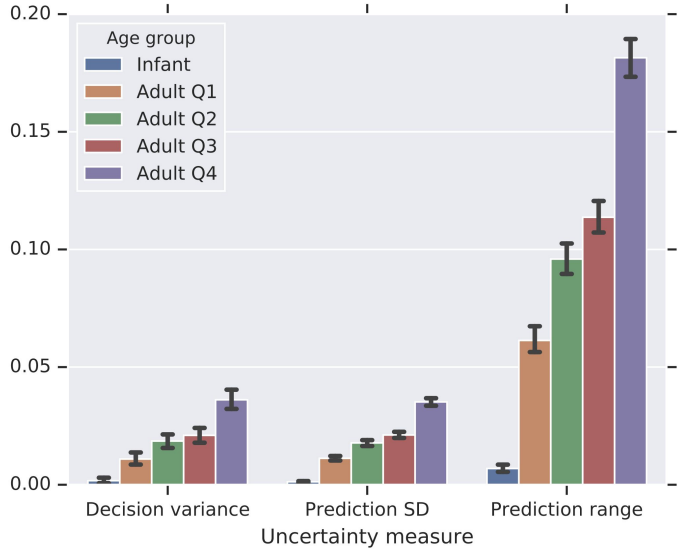
predicted probability of correctness

observed frequency of correctness

[See our NeurIPS'2020 tutorial for background]

Why Reliable Deep Learning?





Test input may not belong to one of the K training classes.

Need to be able to say "none-of-the-above".

Prediction quality may differ drastically across subgroups.

[See our NeurIPS'2020 tutorial for background]

Landscape of Tasks

Google AI Brain Team

Selective Prediction

- Use uncertainty to decide when to trust the model prediction.
- *Google Health, Assistant, Large model safety*

Open Set Recognition

- Test inputs may not belong to one of the existing training classes.
- *Google Health, Assistant*

Sequential Decision-Making

- Active learning
  - *ALFA, Image Understanding*
- Bayesian optimization
  - *Vizier, Sequin*
- Exploration in Bandits / RL
  - *YouTube, RecSys*

Robustness

- Ensure models generalize across distribution shifts.
- Covariate shift
  - *Waymo, Ads*
- *Subpopulation shift (ex. fairness)*

Data Uncertainty

- Assess classification over multiple ratings, across noise and ambiguity
  - *Web-image data*
  - *Content moderation systems*

Adaptation

- Assess ability to adapt to new tasks: ex. quickly, not forgetting.
- Few-shot learning
  - *ImageNet, GLUE*
- Continual learning

The **uncertainty-robustness frontier**

Google AI
Brain Team

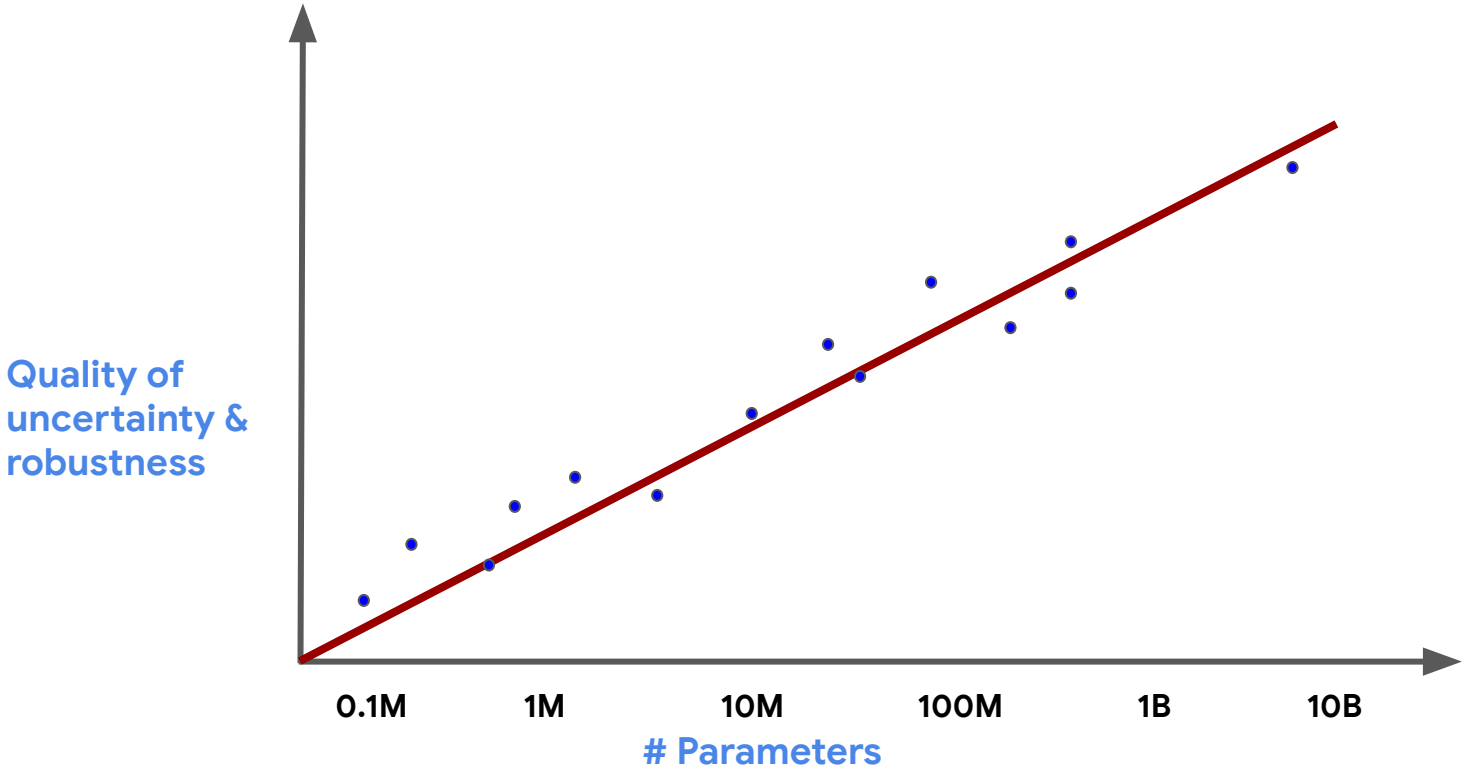
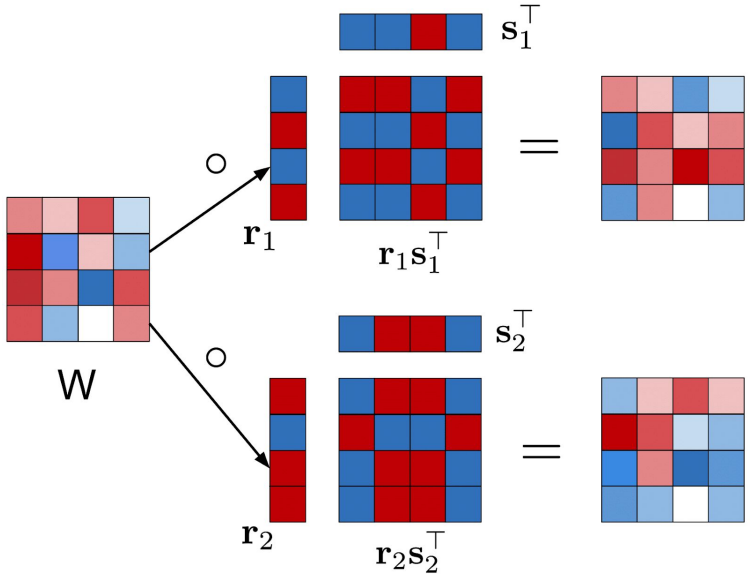
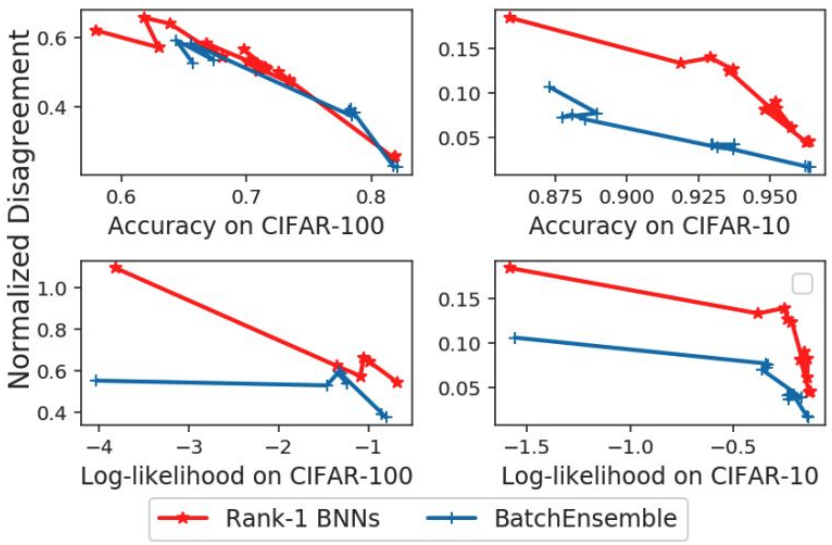


Image source: NeurIPS 2020 tutorial on Uncertainty & Robustness

Ensembles improve uncertainty-robustness across the frontier.

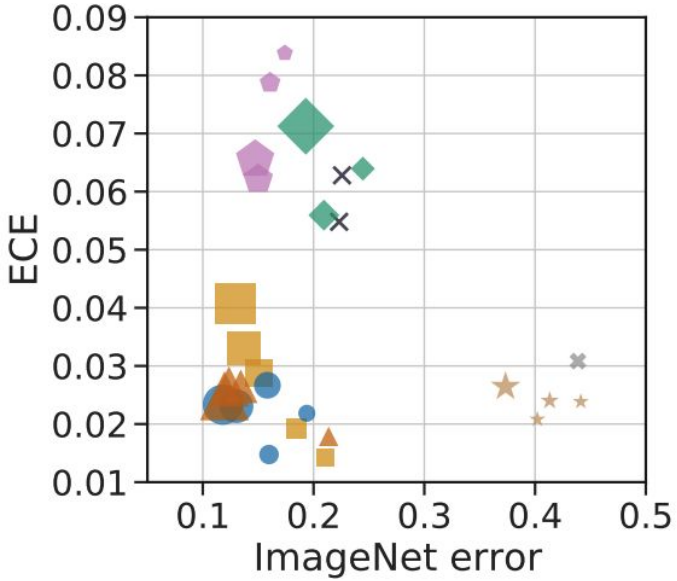


[Wen+ 2020]

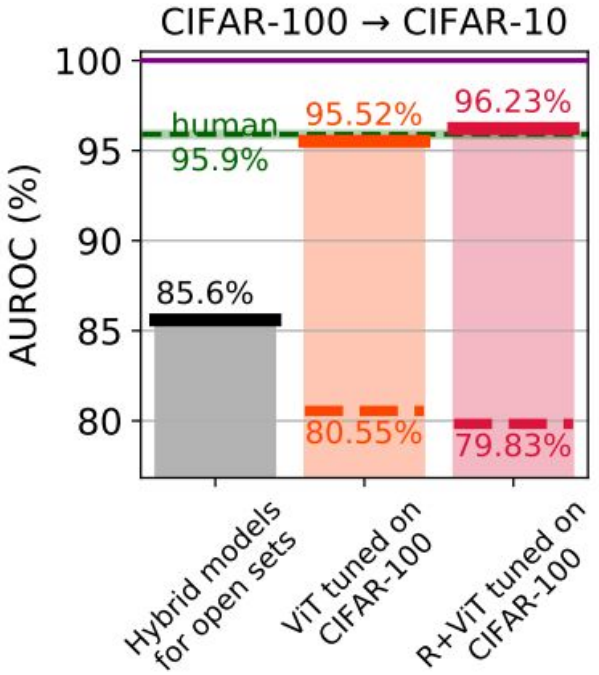


[Dusenberry+ 2020]

Large models are SOTA in calibration & OOD detection.

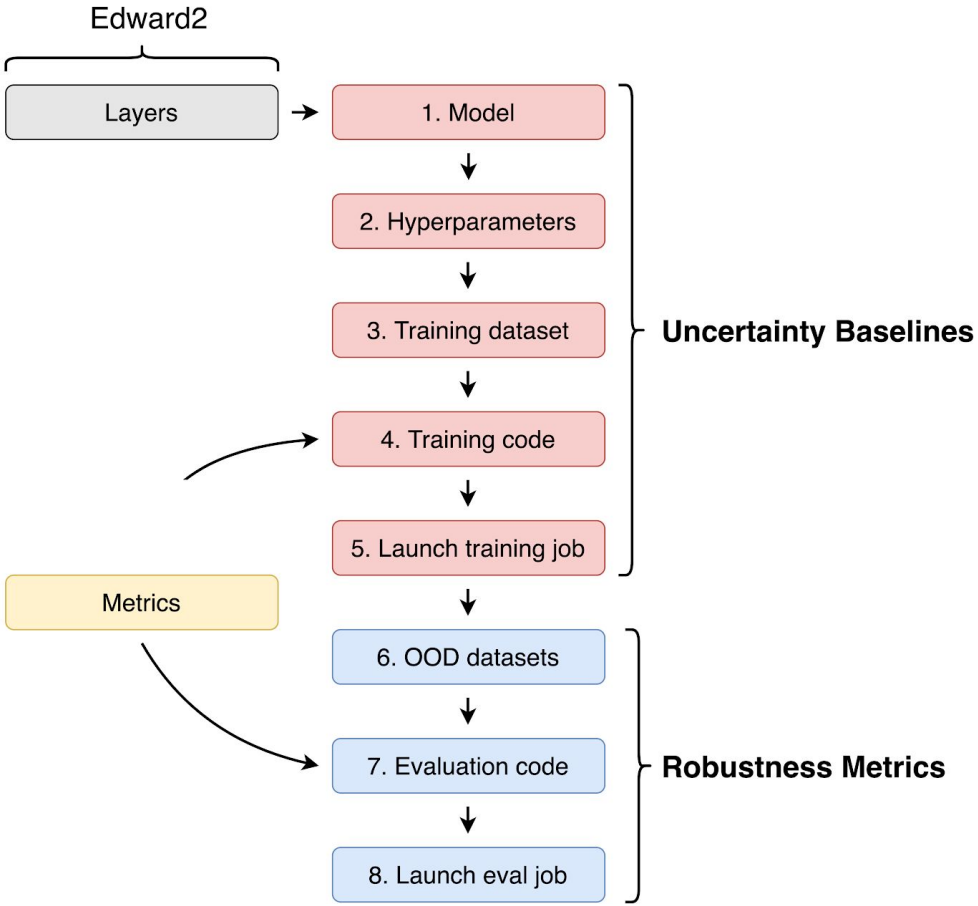


[Minderer+ 2021]



[Fort+ 2021]

Infrastructure



Goal. Provide an experiment template for researchers, with separate modules to use as needed.

Supports JAX, TF, & Pytorch!

Edward2

Language: Random Variables

Edward's language augments an existing ecosystem with random variables. Each random variable $\mathbf{x}$ is associated to an array $\mathbf{x}^*$, $\mathbf{x}^* \sim p(\mathbf{x} \mid \theta^*)$.

```tensorflow
import edward2.tensorflow as ed

# univariate normal
ed.Normal(loc=tf.constant(0.0), scale=tf.constant(1.0))
# vector of 5 univariate normals
ed.Normal(loc=tf.zeros(5), scale=tf.ones(5))
# 2 x 3 matrix of Exponentials
ed.Exponential(rate=tf.ones([2, 3]))
```

TensorFlow (+Distributions)

```numpy
import edward2.numpy as ed

# univariate normal
ed.norm.rvs(loc=0., scale=1.0)
# vector of 5 univariate normals
ed.norm.rvs(loc=np.zeros(5), scale=np.ones(5))
# 2 x 3 matrix of Exponentials
ed.expon.rvs(rate=np.ones([2, 3]))
```

NumPy (+SciPy)

[Tran+ 2017, Tran+ 2018]

Language: Random Variables

Edward2 reifies any computable probability distribution as a Python function. Inputs to the function represent values the distribution conditions on.

```
import edward2.numpy as ed

def linear_model(X):
  beta = ed.norm.rvs(loc=0., scale=0.1, size=X.shape[1])
  loc = np.einsum('ij,j->i', X, beta)
  y = ed.norm.rvs(loc=loc, scale=1.)
  return y

log_joint_fn = ed.make_log_joint_fn(linear_model)
```

Tracing—a tool in automatic differentiation—lets us manipulate the computation.

The neural network language

Neural networks decompose as a composition of **layers**.

Layers represent parameterized functions over (lists of) real tensors. They include:

- **Initializers**. function: shape → array
- **Regularizers**. function: weights → scalar

There are also higher-order layers (ex. `Sequential`) and simple ops (ex. `Add`).

Uncertainty in neural networks

We extend neural networks with random variables as part of any state:

1. Bayesian neural network layers (weights / units)
2. Gaussian process layers (function)
3. Stochastic output layers (output)
4. Reversible layers (input)

Similar ideas in GPflux (2021).

[Tran+ 2019]

Google AI
Brain Team

```
batch_size = 256
units = 512
features, labels = load_dataset(batch_size)
num_features = features.shape[-1]

def model(features):
  qwx = ed.Normal(loc=tf.zeros([num_features, 4 * units]), scale=1., name="wx")
  qwh = ed.Normal(loc=tf.zeros([units, 4 * units]), scale=1., name="wh")
  bias = ed.Normal(loc=tf.zeros([4 * units]), scale=1., name="wh")
  state = tf.ones(units)
  output_layer = tf.keras.layers.Dense(labels.shape[-1])
  logits = []
  for t in range(features.shape[1]):
    z = tf.matmul(features[:, t], qwx) + tf.matmul(net, qwh) + bias
    i, f, o, u = tf.split(z, 4, axis=-1)
    i = tf.sigmoid(i)
    f = tf.sigmoid(f + 1.0)
    o = tf.sigmoid(o)
    u = tf.tanh(u)
    state = f * state + i * u
    net = output * tf.tanh(state)
    logits.append(output_layer(net))
  return ed.Categorical(logits=logits)

def variational():
  wx_loc = tf.get_variable("kernel/input/loc", [num_features, 4 * units], dtype=tf.float32)
  wx_log_scale = tf.get_variable("kernel/input/log_scale", [num_features, 4 * units], dtype=tf.float32)
  wh_loc = tf.get_variable("kernel/hidden/loc", [units, 4 * units], dtype=tf.float32)
  wh_log_scale = tf.get_variable("kernel/hidden/log_scale", [units, 4 * units], dtype=tf.float32)
  bias_loc = tf.get_variable("bias/loc", [4 * units], dtype=tf.float32)
  bias_log_scale = tf.get_variable("bias/loc", [4 * units], dtype=tf.float32)
  qwx = ed.Normal(loc=wx_loc, scale=tf.exp(wx_log_scale), name="qwx")
  qwh = ed.Normal(loc=wh_loc, scale=tf.exp(wh_log_scale), name="qwh")
  qbias = ed.Normal(loc=bias_loc, scale=tf.exp(bias_log_scale))
  return qwx, qwh, qbias

qwx, qwh, bias = variational()
with ed.tape() as tape:
  with ed.trace(ed.set_value(wx=qwx, wh=qwh, bias=qbias)):
    predictions = model(features)

nll = -tf.reduce_mean(predictions.distribution.log_prob(labels))
kl = qwx.distribution.kl_divergence(tape["wx"].distribution)
kl += qwh.distribution.kl_divergence(tape["wh"].distribution)
kl += qbias.distribution.kl_divergence(tape["bias"].distribution)
kl /= dataset_size
loss = nll + kl
train_op = tf.train.AdamOptimizer().minimize(loss)
```

Bayesian LSTM

```
batch_size = 256
features, labels = load_dataset(batch_size)
lstm = layers.LSTMCellReparameterization(512)
output_layer = tf.keras.layers.Dense(labels.shape[-1])
state = lstm.get_initial_state(features)
nll = 0.
for t in range(features.shape[1]):
  net, state = lstm(features[:, t], state)
  logits = output_layer(net)
  nll += tf.losses.softmax_cross_entropy(
      onehot_labels=labels[:, t], logits=logits)

kl = sum(lstm.losses) / dataset_size
loss = nll + kl
optimizer = tf.train.AdamOptimizer()
train_op = optimizer.minimize(loss)
```

Bayesian LSTM in Edward2

Google AI
Brain Team

```
def conv_block(inputs, kernel_size, filters, strides=(2, 2)):
  filters1, filters2, filters3 = filters
  x = layers.Conv2D(filters1, (1, 1),
      strides=strides)(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2D(filters2, kernel\_size,
      padding='SAME')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2D(filters3, (1, 1))(x)
  x = tf.keras.layers.BatchNormalization()(x)
  shortcut = layers.Conv2D(filters3, (1,1),
      strides=strides)(inputs)
  shortcut = tf.keras.layers.BatchNormalization()(shortcut)
  x = tf.keras.layers.add([x, shortcut])
  x = tf.keras.layers.Activation('relu')(x)
  return x

def identity_block(inputs, kernel_size, filters):
  filters1, filters2, filters3 = filters
  x = layers.Conv2D(filters1,(1,1))(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2D(filters2, kernel\_size,
      padding='SAME')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2D(filters3, (1,1))(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.add([x, inputs])
  x = tf.keras.layers.Activation('relu')(x)
  return x
```

```
def build_bayesian_resnet50(input_shape=None,
                            num_classes=1000):
  inputs = tf.keras.layers.Input(shape=input_shape,
                                 dtype='float32')
  x = tf.keras.layers.ZeroPadding2D((3, 3))(inputs)
  x = layers.Conv2D(64, (7, 7),
      strides=(2, 2), padding='VALID')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = tf.keras.layers.ZeroPadding2D((1, 1))(x)
  x = tf.keras.layers.MaxPooling2D((3,3), strides=(2,2))(x)
  x = conv_block(x, 3, [64, 64, 256], strides=(1, 1))
  x = identity_block(x, 3, [64, 64, 256])
  x = identity_block(x, 3, [64, 64, 256])
  x = conv_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = conv_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = conv_block(x, 3, [512, 512, 2048])
  x = identity_block(x, 3, [512, 512, 2048])
  x = identity_block(x, 3, [512, 512, 2048])
  x = tf.keras.layers.GlobalAveragePooling2D()(x)
  x = layers.Dense(num\_classes)(x)
  model = models.Model(inputs, x, name='resnet50')
  return model
```

ResNet-50

Google AI
Brain Team

```python
def conv_block(inputs, kernel_size, filters, strides=(2, 2)):
  filters1, filters2, filters3 = filters
  x = layers.Conv2DVariationalDropout(filters1, (1, 1),
      strides=strides)(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DVariationalDropout(filters2, kernel_size,
      padding='SAME')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DVariationalDropout(filters3, (1, 1))(x)
  x = tf.keras.layers.BatchNormalization()(x)
  shortcut = layers.Conv2DVariationalDropout(filters3, (1,1),
      strides=strides)(inputs)
  shortcut = tf.keras.layers.BatchNormalization()(shortcut)
  x = tf.keras.layers.add([x, shortcut])
  x = tf.keras.layers.Activation('relu')(x)
  return x

def identity_block(inputs, kernel_size, filters):
  filters1, filters2, filters3 = filters
  x = layers.Conv2DVariationalDropout(filters1,(1,1))(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DVariationalDropout(filters2, kernel_size,
      padding='SAME')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DVariationalDropout(filters3, (1,1))(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.add([x, inputs])
  x = tf.keras.layers.Activation('relu')(x)
  return x
```

```python
def build_bayesian_resnet50(input_shape=None,
                            num_classes=1000):
  inputs = tf.keras.layers.Input(shape=input_shape,
                                 dtype='float32')
  x = tf.keras.layers.ZeroPadding2D((3, 3))(inputs)
  x = layers.Conv2DVariationalDropout(64, (7, 7),
      strides=(2, 2), padding='VALID')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = tf.keras.layers.ZeroPadding2D((1, 1))(x)
  x = tf.keras.layers.MaxPooling2D((3,3), strides=(2,2))(x)
  x = conv_block(x, 3, [64, 64, 256], strides=(1, 1))
  x = identity_block(x, 3, [64, 64, 256])
  x = identity_block(x, 3, [64, 64, 256])
  x = conv_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = identity_block(x, 3, [128, 128, 512])
  x = conv_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = identity_block(x, 3, [256, 256, 1024])
  x = conv_block(x, 3, [512, 512, 2048])
  x = identity_block(x, 3, [512, 512, 2048])
  x = identity_block(x, 3, [512, 512, 2048])
  x = tf.keras.layers.GlobalAveragePooling2D()(x)
  x = layers.DenseReparameterization(num_classes)(x)
  model = models.Model(inputs, x, name='resnet50')
  return model
```

Bayesian ResNet-50 in Edward2

Uncertainty Baselines

Go to slide deck here.

Computer Science > Machine Learning

[Submitted on 7 Jun 2021]

Uncertainty Baselines: Benchmarks for Uncertainty & Robustness in Deep Learning

Zachary Nado, Neil Band, Mark Collier, Josip Djolonga, Michael W. Dusenberry, Sebastian Farquhar, Angelos Filos, Marton Havasi, Rodolphe Jenatton, Ghassen Jerfel, Jeremiah Liu, Zelda Mariet, Jeremy Nixon, Shreyas Padhy, Jie Ren, Tim G. J. Rudner, Yeming Wen, Florian Wenzel, Kevin Murphy, D. Sculley, Balaji Lakshminarayanan, Jasper Snoek, Yarin Gal, Dustin Tran

High-quality estimates of uncertainty and robustness are crucial for numerous real-world applications, especially for deep learning which underlies many deployed ML systems. The ability to compare techniques for improving these estimates is therefore very important for research and practice alike. Yet, competitive comparisons of methods are often lacking due to a range of reasons, including: compute availability for extensive tuning, incorporation of sufficiently many baselines, and concrete documentation for reproducibility. In this paper we introduce Uncertainty Baselines: high-quality implementations of standard and state-of-the-art deep learning methods on a variety of tasks. As of this writing, the collection spans 19 methods across 9 tasks, each with at least 5 metrics. Each baseline is a self-contained experiment pipeline with easily reusable and extendable components. Our goal is to provide immediate starting points for experimentation with new methods or applications. Additionally we provide model checkpoints, experiment outputs as Python notebooks, and leaderboards for comparing results. Code available at this https URL.

https://github.com/google/uncertainty-baselines

What is it?

Uncertainty Baselines is:

- Common dataset loaders
- Common models and variants involved in uncertainty research
- Training scripts that use these datasets/models

Our goals is for users to:

- Reproduce our uncertainty research for their benchmarking
- Build off of our work in novel research

Who uses it?

1. Simple and Principled Uncertainty Estimation with Deterministic Deep Learning via Distance Awareness
2. Revisiting One-vs-All Classifiers for Predictive Uncertainty and Out-of-Distribution Detection in Neural Networks
3. Measuring Calibration in Deep Learning
4. Refining the variational posterior through iterative optimization
5. Prediction-Time Batch Normalization for Robustness under Covariate Shift
6. Distilling Ensembles Improves Uncertainty Estimates
7. A Simple Fix to Mahalanobis Distance for Improving Near-OOD Detection
8. Exploring the Uncertainty Properties of Neural Networks' Implicit Priors in the Infinite-Width Limit
9. Measuring and Improving Model-Moderator Collaboration using Uncertainty Estimation
10. Soft Calibration Objectives for Neural Networks
11. DEUP: Direct Epistemic Uncertainty Prediction
12. Neural networks with late-phase weights
13. On the Practicality of Deterministic Epistemic Uncertainty
14. FreeTickets: Accurate, Robust and Efficient Deep Ensemble by Training with Dynamic Sparsity

Who uses it?

Several Google-internal product launches have used Uncertainty Baselines for experimenting with ideas on model performance under distribution shift

Testing

Testing deep learning code is hard!

- End-to-end tests can be flaky due to non-determinism, driver updates, etc.
- Final convergence tests can take hours/days to complete

Testing

We have wide unit test coverage of all our datasets and models:

- Checks shapes, dtypes, and min/max values of outputs
- Pytype helps catch some mismatches during development too

Quick training tests that run for several steps also useful

- Check loss after several weight updates to ensure pipeline runs end-to-end

Testing

Determinism is key for having useful testing and debugging

Jax and TF have splittable/foldable PRNGs that can be passed to all random code:

- `jax.PRNGKey` (docs)
- `tf.random.experimental.stateless_{fold_in,split}`

Our CIFAR deterministic TF baseline uses this for true deterministic!*

*given the same hardware and versions of hardware drivers

Frameworks

Many hot takes on which framework is best!

- Google obviously has a lot of tech built in TF
- PyTorch is a popular favorite externally
- Jax is quickly gaining popularity, especially in Google-adjacent circles

Frameworks

1. Uncertainty Baselines started in TensorFlow 2
2. External collaborators added PyTorch model implementations
    a. Easier than rewriting the model library in TF or Jax
3. We have started added more Jax models (JFT)

All use same tf.data pipelines, either directly via tf.Tensors or converting to np.array on the CPU before feeding into accelerators

Project structure philosophy

Most codebases agree on having standalone models/datasets that are reusable.

Usually train/eval loops where the differences happen.

- Totally standalone scripts, reimplement train/eval code for every workload
- Some shared train/eval code for common use cases
- Abstract away reusable APIs for train/eval (tf.Estimator)

Project structure philosophy

Most codebases agree on having standalone models/datasets that are reusable.

Usually train/eval loops where the differences happen.

- **Totally standalone scripts, reimplement train/eval code for every workload**
- Some shared train/eval code for common use cases
- Abstract away reusable APIs for train/eval (tf.Estimator)

Important for codebases like Uncertainty Baselines where have widely different train/eval loops depending on the experiment!

Project structure philosophy

Totally standalone scripts, reimplement train/eval code for every workload

Pros

- Zero dependencies between experiments
- Trivial to take a single script and reproduce results, less setup
- Cleaner to fork the exact experiments you want
- Easier to understand full end-to-end train/eval

Cons

- Repeated train code
- More work to add codebase-wide upgrades
- More tests to check all experiments working

Project structure

- baselines/
  - cifar/
    - deterministic.py
    - experiments/deterministic_tune.py
  - diabetic_retinopathy_detection/
  - imagenet/
  - jft/
- uncertainty_baselines/
  - halton.py
  - datasets/
    - cifar.py
    - toxic_comments.py
  - models/
    - resnet50_variational.py
    - vit_gp.py

Project structure

- baselines/
  - cifar/
    - deterministic.py
    - experiments/deterministic_tune.py
  - diabetic_retinopathy_detection/
  - imagenet/
  - jft/
- uncertainty_baselines/
  - halton.py
  - datasets/
    - cifar.py
    - toxic_comments.py
  - models/
    - resnet50_variational.py
    - vit_gp.py

baselines/ contains training scripts and experiment configs, organized by workload

Project structure

- baselines/
  - cifar/
    - deterministic.py
    - experiments/deterministic_tune.py
  - diabetic_retinopathy_detection/
  - imagenet/
  - jft/
- uncertainty_baselines/
  - halton.py
  - datasets/
    - cifar.py
    - toxic_comments.py
  - models/
    - resnet50_variational.py
    - vit_gp.py

halton.py contains shuffled quasi-random sequence generator for reproducible hyperparameter tuning

Google AI
Brain Team

Project structure

- baselines/
  - cifar/
    - deterministic.py
    - experiments/deterministic_tune.py
  - diabetic_retinopathy_detection/
  - imagenet/
  - jft/
- uncertainty_baselines/
  - halton.py
  - datasets/
    - cifar.py
    - toxic_comments.py
  - models/
    - resnet50_variational.py
    - vit_gp.py

datasets/ contains tf.data pipelines that produce tf.Tensor or np.array batches of data

Project structure

- baselines/
  - cifar/
    - deterministic.py
    - experiments/deterministic_tune.py
  - diabetic_retinopathy_detection/
  - imagenet/
  - jft/
- uncertainty_baselines/
  - halton.py
  - datasets/
    - cifar.py
    - toxic_comments.py
  - models/
    - resnet50_variational.py
    - vit_gp.py

models/ contains tf.keras, jax Flax, and PyTorch models, with many variants containing robustness-improving layers

Datasets

`tf.data` pipelines supply our data

- Many are hosted in TensorFlow Datasets
- Also have an API for implementing custom or local datasets
- Allows for preprocessing parallelism on CPU

Abstract base class to define API in `datasets/base.py`

- Only requires a preprocessing fn to be defined for subclasses

Datasets

`tf.data` pipelines supply our data

Dataset builder object sets up dataset pipeline with all required hyperparameters:

```
train_builder = ub.datasets.ImageNetDataset(
    split=tfds.Split.TRAIN,
    mixup_params=mixup_params,
    validation_percent=1.0 - FLAGS.train_proportion)
train_ds = train_builder.load(batch_size=bs, strategy=strategy)
```

Datasets

`tf.data` pipelines supply our data

Dataset builder object sets up dataset pipeline with all required hyperparameters:

```
train_builder = ub.datasets.ImageNetDataset(
    split=tfds.Split.TRAIN,
    mixup_params=mixup_params,
    validation_percent=1.0 - FLAGS.train_proportion)
train_ds = train_builder.load(batch_size=bs, strategy=strategy)
```

For TF multi-host training, need to pass in the DistributionStrategy to properly shard the dataset across hosts

Datasets

`tf.data` pipelines supply our data

Dataset builder object sets up dataset pipeline with all required hyperparameters:

```
train_builder = ub.datasets.ImageNetDataset(
    split=tfds.Split.TRAIN,
    mixup_params=mixup_params,
    validation_percent=1.0 - FLAGS.train_proportion)
train_ds = train_builder.load(batch_size=bs, strategy=strategy)
train_iterator = iter(train_ds)
for batch in train_iterator:  # np.array of shape [B, ...]
  ...
```

Models

Making a framework-agnostic model API seems like more hassle than worth:

- Device placement, graph compilation, state tracking, autodiff

Instead one function per `.py` file that returns tf.keras, PyTorch, or Flax model

Metrics

Uncertainty Baselines does not define our own metrics.

Instead use Robustness Metrics, another Google Brain codebase

- Numerous framework-agnostic (all numpy) metrics
- Eval code for running on numerous test sets

General API (defined in their base.py) for a metrics is to define:

- **`add_predictions`** and/or **`add_batch`** to include model outputs in result
- **`result`** to compute final metric value

Experiment configs

Experiment configuration files contain all the hyperparameters, tuning search spaces, and/or cloud machine info to reproduce an experiment. Examples

- .sh with command to launch experiment
- absl FLAGS file with all command line flags used
- README files with instructions on how to reproduce
- ???

Critically important to check these into version control!

Experiment configs

Uncertainty Baselines uses `.py` files that return `ml_collections.ConfigDict`, a lightweight wrapper around a Python `dict` (example)

- Python file allows for parameterization of experiment configs
- These `ConfigDicts` consumed by our (soon to be open-sourced) cloud launcher, passed to train script as various absl FLAG values

Hyperparameters

Another area besides train/eval code that codebases differ:

How are hyperparameters managed?

- Python dict
- Dependency injection framework
- Global FLAGS throughout code
- Individual kwargs
- Custom hparams objects
- ???

Hyperparameters

I asked Twitter and got answers involving all of these approaches!

Hyperparameters

Uncertainty Baselines uses individual absl FLAGS for each hyperparameter, and passes them around our code as individual keyword arguments.

Pros

- Easier to understand for new users trying to rerun or fork our code
- Very clear how each value is treated

Cons

- Very verbose, but standalone train/eval scripts means no nested layers of library functions making this painful

Thank you!

You can clone or pip install our code to try it,
or open an issue or PR if you want to contribute!