Bayesian Layers: A Module for Neural Network Uncertainty

Dustin Tran[*] **Michael W. Dusenberry**[*][*] **Mark van der Wilk**[†] **Danijar Hafner**[*]

[*]Google Brain, [†]Prowler.io

Abstract

We describe Bayesian Layers, a module designed for fast experimentation with neural network uncertainty. It extends neural network libraries with layers capturing uncertainty over weights (Bayesian neural nets), pre-activation units (dropout), activations ("stochastic output layers"), and the function itself (Gaussian processes). With reversible layers, one can also propagate uncertainty from input to output such as for flow-based distributions and constant-memory backpropagation. Bayesian Layers are a drop-in replacement for other layers, maintaining core features that one typically desires for experimentation. As demonstration, we fit a 10-billion parameter "Bayesian Transformer" on 512 TPUv2 cores, which replaces attention layers with their Bayesian counterpart.[1]

```
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)
loss = nll + kl
optimizer = tf.train.AdamOptimizer()
train_op = optimizer.minimize(loss)
```

Figure 1: Bayesian RNN (Fortunato et al., 2017). Bayesian Layers integrate easily into existing workflows.

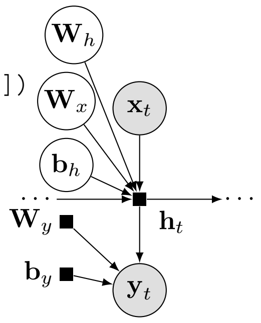

Figure 2: Graphical model depiction. Default arguments specify learnable distributions over the LSTM's weights and biases; we apply a deterministic output layer.

1 Introduction

The rise of AI accelerators such as TPUs lets us utilize computation with 10^{16} FLOP/s and 4 TB of memory distributed across hundreds of processors (Jouppi et al., 2017). This lets us fit probabilistic models at many orders of magnitude larger than state of the art. In particular, we are motivated by research on priors and algorithms for Bayesian neural networks (e.g., Wen et al., 2018; Hafner et al., 2018), scaling up Gaussian processes (e.g., Salimbeni and Deisenroth, 2017; John and Hensman, 2018), and expressive distributions via invertible functions (e.g., Rezende and Mohamed, 2015).

[*]Work done as a Google AI Resident.

[1]All code is available at `https://github.com/tensorflow/tensor2tensor`. Namespaces: `import tensorflow as tf; ed=edward2`. Code snippets assume `tensorflow==1.12.0`.

```
outputs = layers.DenseReparameterization(512, activation=tf.nn.relu)(inputs)
outputs = layers.DenseFlipout(512, activation=tf.nn.relu)(inputs)
outputs = layers.DenseQuadrature(512, activation=tf.nn.relu)(inputs)
```

Figure 3: Bayesian feedforward layer with different estimators, including reparameterization (Kingma and Welling, 2014), Flipout (Wen et al., 2018), and numerical quadrature.

Unfortunately, while research with these methods are not limited by hardware, they are limited by software. This paper describes Bayesian Layers, an extension of neural network libraries which contributes one idea: instead of only deterministic functions as "layers", enable distributions over functions. Our implementation extends Keras in TensorFlow (Chollet, 2016) and uses Edward2 (Tran et al., 2018) to operate with random variables.

1.1 Related Work

There have been many software developments for distributions over functions. Our work takes classic inspiration from Radford Neal's software in 1995 to enable flexible modeling with both Bayesian neural nets and GPs (Neal, 1995). Modern software typically focuses on only one of these directions. For Bayesian neural nets, researchers have commonly coupled variational sampling in neural net layers (e.g., code and algorithms from Gal and Ghahramani (2016); Louizos and Welling (2017)). For Gaussian processes, there have been significant developments in libraries (Rasmussen and Nickisch, 2010; GPy, 2012; Vanhatalo et al., 2013; Matthews et al., 2017; Al-Shedivat et al., 2017; Gardner et al., 2018). Perhaps most similar to our work, Aboleth (Aboleth Developers, 2017) features variational BNNs and random feature approximations for GPs. Aside from API differences from all these works, our work tries to revive the spirit of enabling any function with uncertainty—whether that be, e.g., in the weights, activations, or the entire function—and to do so in a manner compatible with scalable deep learning ecosystems.[2]

2 Bayesian Layers

In neural network libraries, architectures decompose as a composition of "layer" objects as the core building block (Collobert et al., 2011; Al-Rfou et al., 2016; Jia et al., 2014; Chollet, 2016; Chen et al., 2015; Abadi et al., 2015; S. and N., 2016). These layers capture both the parameters and computation of a mathematical function into a programmable class. In our work, we extend layers to capture "distributions over functions", which we describe as a layer with uncertainty about some state in its computation—be it uncertainty in the weights, pre-activation units, activations, or the entire function. Each sample from the distribution instantiates a different function, e.g., a layer with a different weight configuration.

2.1 Bayesian Neural Network Layers

The Bayesian extension of any deterministic layer is to place a prior distribution over its weights and biases. These layers require several considerations. Figure 1 implements a Bayesian RNN.

Computing the integral We need to compute often-intractable integrals over weights and biases θ. We focus on two cases, the variational objective $\text{ELBO}(\theta) = \int q(\theta) \log p(\text{outputs} \mid \text{layer}_\theta(\text{inputs}))\, \mathrm{d}\theta - \text{KL}\left[q(\theta) \,\|\, p(\theta)\right]$, and the approximate predictive distribution $q(\text{outputs} \mid \text{inputs}) = \int q(\theta) p(\text{outputs} \mid \text{layer}_\theta(\text{inputs}))\, \mathrm{d}\theta$. To enable different methods to estimate these integrals, we implement each estimator as its own Layer (Figure 3). The same Bayesian neural net can use entirely different computational graphs depending on the estimation (and therefore entirely different code). For example, sampling from $q(\theta)$ with reparameterization and running the deterministic layer computation is a generic way to evaluate integrals (Kingma and Welling, 2014).

Signature We'd like to have the Bayesian extension of a deterministic layer retain its mandatory constructor arguments as well as Tensor-in Tensor-out call signature. This avoids cognitive overhead, letting one easily swap layers (Figure 4).

[2] Historically, an older design of Bayesian Layers was implemented in TensorFlow Probability (TensorFlow Probability Developers, 2017); the current design exists in Tensor2Tensor (Vaswani et al., 2018). In the future as the API stabilizes, we hope Bayesian Layer ideas may make it into core TensorFlow Keras and/or TensorFlow Probability.

```
if FLAGS.be_bayesian:
  Conv2D = layers.Conv2DReparameterization
else:
  Conv2D = tf.keras.layers.Conv2D

model = tf.keras.Sequential([
    Conv2D(32, kernel_size=5, strides=1, padding='SAME'),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Activation('relu'),
    Conv2D(32, kernel_size=5, strides=2, padding='SAME'),
    tf.keras.layers.BatchNormalization(),
    ...
])
```

Figure 4: Bayesian Layers are drop-in replacements for their deterministic counterparts.

Distributions over parameters To specify distributions, a natural idea is to overload the existing parameter initialization arguments in a Layer's constructor, such as `kernel_initializer` and `bias_initializer` in Keras. These arguments are extended to accept callables that take metadata such as input shape and return a distribution over the parameter. Distribution initializers may carry trainable parameters, each with their own initializers (pointwise or distributional). The default initializer represents a trainable approximate posterior in a variational inference scheme.

For the distribution abstraction, we use Edward `RandomVariables` (Tran et al., 2018). They are Tensors augmented with distribution methods such as `sample` and `log_prob`; by default, numerical ops operate on its sample Tensor. Layers perform forward passes using deterministic ops and the `RandomVariables`.

Distribution regularizers The variational training objective requires the evaluation of a KL term, which penalizes deviations of the learned $q(\theta)$ from the prior $p(\theta)$. Similar to distribution initializers, we overload the existing parameter regularization arguments in a layer's constructor, such as `kernel_regularizer` and `bias_regularizer` in Keras. These arguments are extended to accept callables that take in the kernel or bias `RandomVariables` and return a scalar Tensor. By default, we use a KL divergence toward the standard normal distribution, which represents the penalty term common in variational Bayesian neural network training.

Explicitly returning regularizers in a Layer's call ruins composability (see Signature above). Therefore Bayesian layers, like their deterministic counterparts, side-effect the computation: one queries an attribute to access any regularizers for, e.g., the loss function. Figure 1 implements a Bayesian RNN; Appendix A implements a Bayesian CNN (ResNet-50).

2.2 Gaussian Process Layers

As opposed to representing distributions over functions through the weights, Gaussian processes represent distributions over functions by specifying the value of the function at different inputs. Recent advances have made Gaussian process inference computationally similar to Bayesian neural networks (Hensman et al., 2013). We only require a method to sample the function value at a new input, and evaluate KL regularizers, which allows GPs to be placed in the same framework as above.[3] Figure 5 implements a deep GP.

Signature. Use the same mandatory arguments as their equivalent deterministic layer (e.g., number of units in a layer determine a GP's output dimensionality; a GP's kernel is an optional argument defaulting to squared exponential). Retain the Tensor-in Tensor-out call signature.

Computing the integral. Use a separate class per estimator, such as `GaussianProcess` for exact integration, which is only possible in limited situations; `SparseGaussianProcess` for inducing variable approximations; and `RandomFourierFeatures` for projection approximations.

[3]More broadly, these ideas extend to stochastic processes. For example, we plan to implement a Poisson process layer for scalable point process modeling.

```
batch_size = 256
features, labels = load_spatial_data(batch_size)

model = tf.keras.Sequential([
  tf.keras.layers.Flatten(),  # no spatial knowledge
  layers.SparseGaussianProcess(units=256, num_inducing=512),
  layers.SparseGaussianProcess(units=256, num_inducing=512),
  layers.SparseGaussianProcess(units=10, num_inducing=512),
])
predictions = model(features)
neg_log_likelihood = tf.losses.mean_squared_error(labels=labels,
    predictions=predictions)
kl = sum(model.losses)
loss = neg_log_likelihood + kl
train_op = tf.train.AdamOptimizer().minimize(loss)
```

Figure 5: Three-layer deep GP with variational inference (Salimbeni and Deisenroth, 2017; Damianou and Lawrence, 2013). We apply it for regression given batches of spatial inputs and vector-valued outputs. We flatten inputs to use the default squared exponential kernel; this naturally extends to pass in a more sophisticated kernel function.

```
def build_image_transformer(hparams):
  x = tf.keras.layers.Input(shape=input_shape, dtype='float32')
  x = ChannelEmbedding(hparams.hidden_size)(x)
  x = tf.keras.layers.Reshape([-1, hparams.hidden_size])(x)
  x = tf.pad(x, [[0, 0], [1, 0], [0, 0]])[:, :-1, :])  # shift pixels right
  x = PositionalEmbedding(max_length=128*128*3)(x)
  x = tf.keras.layers.Dropout(0.3)(x)
  for _ in range(hparams.num_layers):
    y = LocalAttention1D(hparams, attention_type='local_mask_right',
                         q_padding='LEFT', kv_padding='LEFT')(x)
    x = LayerNormalization()(tf.keras.layers.Dropout(0.3)(y) + x)
    y = tf.keras.layers.Dense(x, hparams.filter_size, activation=tf.nn.relu)
    y = tf.keras.layers.Dense(hparams.hidden_size, activation=None)(y)
    x = LayerNormalization()(tf.keras.layers.Dropout(0.3)(y) + x)
  x = layers.MixtureofLogistic(3, num_components=5)(x)
  x = layers.Discretize(x)
  model = tf.keras.Model(inputs=inputs, outputs=x, name='ImageTransformer')
  return model

image_transformer = build_image_transformer(hparams)
loss = -tf.reduce_sum(image_transformer(features).distribution.log_prob(features))
train_op = tf.train.AdamOptimizer().minimize(loss)
```

Figure 6: Image Transformer with discretized logistic mixture (Parmar et al., 2018) over 128x128x3 features. We assume layers which don't exist in Keras; functional versions are available in Tensor2Tensor (Vaswani et al., 2018).

Distribution regularizers. By default, include no regularizer for exact GPs, a KL regularizer on inducing outputs for sparse GPs, and a KL regularizer on weights for random projection approximations. These defaults reflect each inference method's standard for training.

2.3 Stochastic Output Layers

In addition to uncertainty over the *mapping* defined by a layer, we may want to simply add stochasticity to the output. These outputs have a tractable distribution, and we often would like to access its properties: for example, maximum likelihood with an autoregressive network whose output is a discretized logistic mixture (Salimans et al., 2017) (Figure 6); or an auto-encoder with stochastic encoders and decoders (Appendix B). To implement stochastic output layers, we simply perform deterministic computations and return a `RandomVariable`. Because `RandomVariable`s are Tensor-like objects, one can operate on them as if they were Tensors: composing stochastic output layers is

```
batch_size = 256
features = load_cifar10(batch_size)

model = tf.keras.Sequential([
  layers.RealNVP(MADE(hidden_dims=[512, 512])),
  layers.RealNVP(MADE(hidden_dims=[512, 512], order='right-to-left')),
  layers.RealNVP(MADE(hidden_dims=[512, 512])),
])
base = ed.Normal(loc=tf.zeros([batch_size, 32*32*3]), scale=1.)
outputs = model(base)
loss = -tf.reduce_sum(outputs.distribution.log_prob(features))
train_op = tf.train.AdamOptimizer().minimize(loss)
```

Figure 7: A flow-based model for image generation (Dinh et al., 2017).

valid. In addition, using such a layer as the last one in a network allows one to compute properties such as a network's entropy or likelihood given data.[4]

2.4 Reversible Layers

With random variables in layers, one can naturally capture invertible neural networks which propagate uncertainty from input to output. In particular, a reversible layer may take a `RandomVariable` as input and return another `RandomVariable` with the same distribution up to a volume change. The layer implements not only a forward pass but also a method `reverse` and optionally `log_det_jacobian`.[5] Figure 7 implements RealNVP (Dinh et al., 2017), which is a reversible layer parameterized by another network (here, MADE (Germain et al., 2015)). These ideas also extend to reversible networks that enable backpropagation without storing intermediate activations in memory during the forward pass (Gomez et al., 2017).

3 Experiments

We implemented a "Bayesian Transformer" for the One-Billion-Word Language Modeling Benchmark (Chelba et al., 2013). Using Mesh TensorFlow (Shazeer et al., 2018), we took a 5-billion parameter Transformer which reports a state-of-the-art perplexity of 23.1. We then augmented the model with priors over the projection matrices by replacing calls to a multihead-attention layer with its Bayesian counterpart (using the Flipout estimator). Figure 10 shows that we can fit models with over 10-billion parameters, utilizing up to 2500 TFLOPs on 512 TPUv2 cores. In attempting these scales, we were able to reach state-of-the-art perplexities at the cost of worse uncertainties than smaller Bayesian Transformers. We identified a number of challenges in scaling up Bayesian neural nets which we leave out of this work, and which we're excited to explore for future research.

4 Limitations

While the framework we laid out tightly integrates deep Bayesian modelling into existing ecosystems, we have deliberately limited our scope—especially compared to fully-fledged probabilistic programming languages. In particular, our layers tie the model specification to the inference algorithm (typically, variational inference). The core assumption of this framework is the modularization of inference per layer. This makes inference procedures which depend on the full parameter space, such as Markov chain Monte Carlo, difficult to fit within the framework. Other inference methods such as EP variants (Bui et al., 2016; Hernández-Lobato and Adams, 2015) could fit if explicit separate representations of prior and approximating distributions are used, in effect, making a layer a special kind of random variable already used in probabilistic programming languages (e.g. Edward).

Acknowledgements. We thank James Hensman and Alex Matthews for motivating discussions around GPs and probabilistic programming; Sergio Guadarrama for the initial idea behind stochastic

[4] In previous figures, we used loss functions such as `mean_squared_error`. With stochastic output layers, we can replace them with a layer returning the likelihood and calling `log_prob`.

[5] We implement `layers.Discretize` this way in Figure 6. It takes a continuous `RandomVariable` as input and returns a transformed variable with probabilities integrated over bins.

output layers; Josh Dillon, Dave Moore, Chris Suter, and Matt Hoffman for API discussions in an older design; and Eugene Brevdo, Martin Wicke, and Kevin Murphy for their feedback.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

Aboleth Developers (2017). Aboleth. https://github.com/data61/aboleth.

Al-Rfou, R., Alain, G., Almahairi, A., Angermueller, C., Bahdanau, D., Ballas, N., Bastien, F., Bayer, J., Belikov, A., Belopolsky, A., Bengio, Y., Bergeron, A., Bergstra, J., Bisson, V., Bleecher Snyder, J., Bouchard, N., Boulanger-Lewandowski, N., Bouthillier, X., de Brébisson, A., Breuleux, O., Carrier, P.-L., Cho, K., Chorowski, J., Christiano, P., Cooijmans, T., Côté, M.-A., Côté, M., Courville, A., Dauphin, Y. N., Delalleau, O., Demouth, J., Desjardins, G., Dieleman, S., Dinh, L., Ducoffe, M., Dumoulin, V., Ebrahimi Kahou, S., Erhan, D., Fan, Z., Firat, O., Germain, M., Glorot, X., Goodfellow, I., Graham, M., Gulcehre, C., Hamel, P., Harlouchet, I., Heng, J.-P., Hidasi, B., Honari, S., Jain, A., Jean, S., Jia, K., Korobov, M., Kulkarni, V., Lamb, A., Lamblin, P., Larsen, E., Laurent, C., Lee, S., Lefrancois, S., Lemieux, S., Léonard, N., Lin, Z., Livezey, J. A., Lorenz, C., Lowin, J., Ma, Q., Manzagol, P.-A., Mastropietro, O., McGibbon, R. T., Memisevic, R., van Merriënboer, B., Michalski, V., Mirza, M., Orlandi, A., Pal, C., Pascanu, R., Pezeshki, M., Raffel, C., Renshaw, D., Rocklin, M., Romero, A., Roth, M., Sadowski, P., Salvatier, J., Savard, F., Schlüter, J., Schulman, J., Schwartz, G., Serban, I. V., Serdyuk, D., Shabanian, S., Simon, E., Spieckermann, S., Subramanyam, S. R., Sygnowski, J., Tanguay, J., van Tulder, G., Turian, J., Urban, S., Vincent, P., Visin, F., de Vries, H., Warde-Farley, D., Webb, D. J., Willson, M., Xu, K., Xue, L., Yao, L., Zhang, S., and Zhang, Y. (2016). Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688*.

Al-Shedivat, M., Wilson, A. G., Saatchi, Y., Hu, Z., and Xing, E. P. (2017). Learning scalable deep kernels with recurrent structure. *Journal of Machine Learning Research*, 18(1).

Bui, T., Hernandez-Lobato, D., Hernandez-Lobato, J., Li, Y., and Turner, R. (2016). Deep gaussian processes for regression using approximate expectation propagation. In Balcan, M. F. and Weinberger, K. Q., editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 1472–1481, New York, New York, USA. PMLR.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., and Koehn, P. (2013). One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005.

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*.

Chollet, F. (2016). Keras. https://github.com/fchollet/keras.

Collobert, R., Kavukcuoglu, K., and Farabet, C. (2011). Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*.

Damianou, A. and Lawrence, N. (2013). Deep gaussian processes. In *Artificial Intelligence and Statistics*, pages 207–215.

Dinh, L., Sohl-Dickstein, J., and Bengio, S. (2017). Density estimation using real nvp. In *International Conference on Learning Representations*.

Fortunato, M., Blundell, C., and Vinyals, O. (2017). Bayesian recurrent neural networks. *arXiv preprint arXiv:1704.02798*.

Gal, Y. and Ghahramani, Z. (2016). Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *international conference on machine learning*, pages 1050–1059.

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., and Wilson, A. G. (2018). Gpytorch: Blackbox matrix-matrix gaussian process inference with gpu acceleration. In *NeurIPS*.

Germain, M., Gregor, K., Murray, I., and Larochelle, H. (2015). Made: Masked autoencoder for distribution estimation. In *International Conference on Machine Learning*, pages 881–889.

Gomez, A. N., Ren, M., Urtasun, R., and Grosse, R. B. (2017). The reversible residual network: Backpropagation without storing activations. In *Neural Information Processing Systems*.

GPy (since 2012). GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`.

Hafner, D., Tran, D., Irpan, A., Lillicrap, T., and Davidson, J. (2018). Reliable uncertainty estimates in deep neural networks using noise contrastive priors. *arXiv preprint*.

Hensman, J., Fusi, N., and Lawrence, N. D. (2013). Gaussian processes for big data. In *Conference on Uncertainty in Artificial Intelligence*.

Hernández-Lobato, J. M. and Adams, R. P. (2015). Probabilistic backpropagation for scalable learning of bayesian neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1861–1869. JMLR.org.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., Guadarrama, S., and Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM.

John, S. T. and Hensman, J. (2018). Large-scale cox process inference using variational fourier features. *arXiv preprint arXiv:1804.01016*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

Kingma, D. P. and Welling, M. (2014). Auto-encoding variational Bayes. In *International Conference on Learning Representations*.

Louizos, C. and Welling, M. (2017). Multiplicative normalizing flows for variational bayesian neural networks. *arXiv preprint arXiv:1703.01961*.

Matthews, A. G. d. G., van der Wilk, M., Nickson, T., Fujii, K., Boukouvalas, A., León-Villagrá, P., Ghahramani, Z., and Hensman, J. (2017). GPflow: A Gaussian process library using TensorFlow. *Journal of Machine Learning Research*, 18(40):1–6.

Neal, R. (1995). Software for flexible bayesian modeling and markov chain sampling. `https://www.cs.toronto.edu/~radford/fbm.software.html`.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, Ł., Shazeer, N., Ku, A., and Tran, D. (2018). Image transformer. In *International Conference on Machine Learning*.

Rasmussen, C. E. and Nickisch, H. (2010). Gaussian processes for machine learning (gpml) toolbox. *Journal of machine learning research*, 11(Nov):3011–3015.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows. In *International Conference on Machine Learning*.

S., G. and N., S. (2016). TensorFlow-Slim: A lightweight library for defining, training and evaluating complex models in TensorFlow.

Salimans, T., Karpathy, A., Chen, X., and Kingma, D. P. (2017). PixelCNN++: Improving the pixelcnn with discretized logistic mixture likelihood and other modifications. *arXiv preprint arXiv:1701.05517*.

Salimbeni, H. and Deisenroth, M. (2017). Doubly stochastic variational inference for deep gaussian processes. In *Advances in Neural Information Processing Systems*, pages 4588–4599.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. (2018). Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*.

TensorFlow Probability Developers (2017). Tensorflow probability. https://github.com/tensorflow/probability.

Tran, D., Hoffman, M. D., Moore, D., Suter, C., Vasudevan, S., Radul, A., Johnson, M., and Saurous, R. A. (2018). Simple, distributed, and accelerated probabilistic programming. In *Neural Information Processing Systems*.

van den Oord, A., Vinyals, O., et al. (2017). Neural discrete representation learning. In *Advances in Neural Information Processing Systems*, pages 6306–6315.

Vanhatalo, J., Riihimäki, J., Hartikainen, J., Jylänki, P., Tolvanen, V., and Vehtari, A. (2013). Gpstuff: Bayesian modeling with gaussian processes. *Journal of Machine Learning Research*, 14(Apr):1175–1179.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. (2018). Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416.

Wen, Y., Vicol, P., Ba, J., Tran, D., and Grosse, R. (2018). Flipout: Efficient pseudo-independent weight perturbations on mini-batches. In *International Conference on Learning Representations*.

A Bayesian ResNet-50

See Figure 8.

B Vector-Quantized Variational Auto-Encoder

See Figure 9.

C Bayesian Transformer

See Figure 10.

```
def conv_block(inputs, kernel_size, filters, strides=(2, 2)):
  filters1, filters2, filters3 = filters
  x = layers.Conv2DFlipout(filters1, (1, 1),
      strides=strides,
      kernel_initializer='he_normal')(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DFlipout(filters2, kernel_size,
      padding='SAME',
      kernel_initializer='he_normal')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DFlipout(filters3, (1, 1),
      kernel_initializer='he_normal')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  shortcut = layers.Conv2DFlipout(filters3, (1, 1),
      strides=strides,
      kernel_initializer='he_normal')(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.Conv2DFlipout(filters1, (1, 1),
      kernel_initializer='he_normal')(inputs)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DFlipout(filters2, kernel_size,
      padding='SAME',
      kernel_initializer='he_normal')(x)
  x = tf.keras.layers.BatchNormalization()(x)
  x = tf.keras.layers.Activation('relu')(x)
  x = layers.Conv2DFlipout(filters3, (1,1),
      kernel_initializer='he_normal')(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.Conv2DFlipout(64, (7, 7),
      strides=(2, 2), padding='VALID',
      kernel_initializer='he_normal')(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.DenseFlipout(num_classes)(x)
  model = models.Model(inputs, x, name='resnet50')
  return model
```

```
bayesian_resnet50 = build_bayesian_resnet50()
logits = bayesian_resnet50(features)
neg_log_likelihood = tf.losses.sparse_softmax_cross_entropy(
    labels=labels, logits=logits, reduction=tf.losses.reduction.MEAN)
kl = sum(bayesian_resnet50.losses)  # include KL penalty which are Layer side-effects
loss = neg_log_likelihood + kl
train_op = tf.train.AdamOptimizer().minimize(loss)

# Alternatively, run the following instead of a manual train_op.
model.compile(optimizer=tf.train.AdamOptimizer(),
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.fit(features, labels, batch_size=32, epochs=5)
```

Figure 8: Bayesian ResNet-50.

```
base_depth = 128

encoder = tf.keras.Sequential([
    tf.keras.layers.Conv2D(base_depth, 5, 1, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2D(base_depth, 5, 2, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2D(2 * base_depth, 5, 1, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2D(2 * base_depth, 5, 2, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2D(4 * latent_size, 7, padding='VALID', activation=tf.nn.relu),
    tf.keras.layers.Flatten(),
    layers.VectorQuantizer(512, name='latent_code'),
])
decoder = tf.keras.Sequential([
    tf.keras.layers.Conv2DTranspose(2 * base_depth, 7, padding='VALID', activation=tf.nn.relu),
    tf.keras.layers.Conv2DTranspose(2 * base_depth, 5, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2DTranspose(2 * base_depth, 5, 2, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2DTranspose(base_depth, 5, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2DTranspose(base_depth, 5, 2, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2DTranspose(base_depth, 5, padding='SAME', activation=tf.nn.relu),
    tf.keras.layers.Conv2D(3, 5, padding='SAME', activation=None),
    layers.Bernoulli(256*256*3, name='image'),  # likelihood
])
encoded_features = encoder(features)
reconstruction = decoder(encoded_features).distribution.log_prob(features)
entropy = encoded_features.distribution.entropy()
loss = reconstruction + entropy
train_op = tf.train.AdamOptimizer().minimize(loss)
```

Figure 9: Vector-quantized variational auto-encoder (van den Oord et al., 2017) for 256x256 ImageNet. VQVAEs assume a uniform prior during training; we use a stochastic encoder.

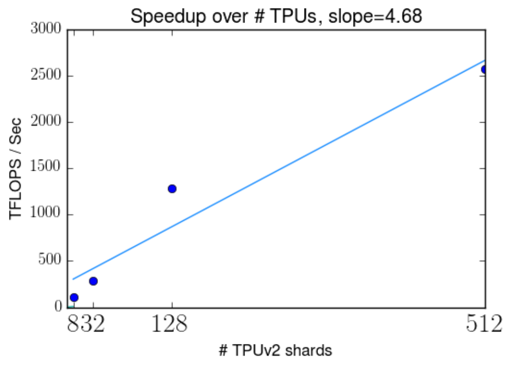

Figure 10: Bayesian Transformer implemented with model parallelism ranging from 8 TPUv2 shards (core) to 512.