TyXe: Pyro-based Bayesian neural nets for PyTorch

Hippolyt Ritter [1 2 †] **Theofanis Karaletsos** [3 †]

Abstract

We introduce TyXe, a Bayesian neural network library built on top of PyTorch and Pyro. Our leading design principle is to cleanly separate architecture, prior, inference and likelihood specification, allowing for a flexible workflow where users can quickly iterate over combinations of these components. In contrast to existing packages TyXe does not implement any layer classes, and instead relies on architectures defined in generic PyTorch code. TyXe then provides modular choices for canonical priors, variational guides, inference techniques, and layer selections for a Bayesian treatment of the specified architecture. Sampling tricks for variance reduction, such as local reparameterization or flipout, are implemented as effect handlers, which can be applied independently of other specifications. We showcase the ease of use of TyXe to explore Bayesian versions of popular models from various libraries: toy regression with a pure PyTorch neural network; large-scale image classification with torchvision ResNets; graph neural networks based on DGL; and Neural Radiance Fields built on top of PyTorch3D. Finally, we provide convenient abstractions for variational continual learning. In all cases the change from a deterministic to a Bayesian neural network comes with minimal modifications to existing code, offering a broad range of researchers and practitioners alike practical access to uncertainty estimation techniques. The library is available at https://github.com/TyXe-BDL/TyXe.

1 Introduction

The surge of interest in deep learning has been fuelled by the availability of agile software packages that enable researchers and practitioners alike to quickly experiment with different architectures for their problem setting (Paszke et al., 2019; Abadi et al., 2016) by providing modular abstractions for automatic differentiation and gradient-based learning. While there has been similarly growing interest in uncertainty estimation for deep neural networks, in particular following the Bayesian paradigm (MacKay, 1992; Neal, 2012), a comparable toolbox of software packages has mostly been missing.

A major barrier of entry for the use of Bayesian Neural Networks (BNNs) is the large overhead in required code and additional mathematical abstractions compared to stochastic maximum likelihood estimation as commonly performed in deep learning. Moreover, BNNs typically have intractable posteriors, necessitating the use of various approximations when performing inference, which depending on the problem may perform better or worse and frequently require complex bespoke implementations. This oftentimes leads to the development of inflexible small libraries or repetitive code creation that can lack essential "tricks of the trade" for performant BNNs, such as appropriate initialization schemes, gradient variance reduction (Kingma et al., 2015; Tran et al., 2018), or may only provide limited inference strategies to compare outcomes. Besides stand-alone libraries such as STAN (Carpenter et al., 2017), various general purpose probabilistic programming packages have been built on top of those deep learning libraries (Pyro (Bingham et al., 2019) for PyTorch, Edward2 (Tran et al., 2018) for Tensorflow), however software linking those to BNNs has only been released recently (Tran et al., 2019) and provides substitutes for Keras' layers (Chollet et al., 2015) to construct BNNs from scratch.

In this work we describe TyXe (Greek: chance), a package linking the expressive computational capabilities of PyTorch with the flexible model and inference design of Pyro (Bingham et al., 2019) in service of providing a simple, agile, and useful abstraction for BNNs targeted at PyTorch practitioners. Specifically, we highlight the following contributions we make through TyXe:

- We provide an intuitive, object-oriented interface that abstracts away Pyro to facilitate turning PyTorch-based neural networks into BNNs with minimal changes to existing code.
- Crucially, our design deviates from prior approaches, e.g. (Tran et al., 2019), to avoid bespoke layer implementations, making TyXe applicable to arbitrary

[1]Meta AI [2]University College London [3]Insitro † work partly done at Uber AI labs and Meta AI.. Correspondence to: Hippolyt Ritter <j.ritter@cs.ucl.ac.uk>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2022.

```
1 net = nn.Sequential(nn.Linear(1, 50), nn.Tanh(), nn.Linear(50, 1))
2 likelihood = tyxe.likelihoods.HomoskedasticGaussian(dataset_size, scale=0.1)
3 prior = tyxe.priors.IIDPrior(dist.Normal(0, 1))
4 guide_factory = tyxe.guides.AutoNormal
5 bnn = tyxe.VariationalBNN(net, prior, likelihood, guide_factory)
```

Listing 1: Bayesian nonlinear regression setup code example in 5 lines. Line 1 is a standard PyTorch neural network definition, line 2 is the likelihood of the data, corresponding to a data loss object. Line 3 sets the prior and line 4 constructs the approximate posterior distribution on the weights. Line 5 finally brings all components together to set up the BNN. For MCMC, the `guide_factory` would be `HMC` or `NUTS` from `pyro.infer.mcmc` and the `BNN` a `tyxe.MCMC_BNN`.

PyTorch architectures.

- We make essential techniques for well-performing BNNs that are missing from Pyro, such as local reparameterization, available as flexible program transformations.
- TyXe is compatible with architectures from libraries both native and non-native to the PyTorch ecosystem, such as torchvision ResNets and DGL graph neural networks, and thus runs on GPU hardware.
- Leveraging TyXe, we show that a Bayesian treatment of PyTorch3d-based Neural Radiance Fields improves their out-of-distribution robustness at a minimal coding overhead.
- Our modular design handily supports variational continual learning through updating the prior to the posterior. Such abstractions are also currently not available in Pyro.

In the following we give an overview of our library, with an initial focus on the API design followed by a range of research settings where TyXe greatly simplifies 'Bayesianizing' an existing deep learning workflow. We provide an in-depth overview of the codebase in Appendix A and discuss specific advancements upon Pyro in Appendix B with a direct comparison to our ResNet example.

2 TyXe by example: non-linear regression

The core components that users interact with in TyXe are our BNN classes. These wrap deterministic PyTorch `nn.Module` neural networks. We then leverage Pyro to formulate a probabilistic model over the neural network parameters, in which we perform approximate inference. There are two primary BNN classes with identical interfaces: `tyxe.VariationalBNN` and `tyxe.MCMC_BNN`. Both offer a unified workflow of constructing a BNN, fitting it to data and then making predictions. A more low-level class, `tyxe.PytorchBNN`, which can act as a drop-in BNN replacement for a `nn.Module` but lacks some of the high-level functionality of the other two classes, will be introduced in Section 4.2. We stress that the former two classes only require using a Pyro optimizer in place of a PyTorch one, while the latter hides Pyro entirely, making its functionality accessible to PyTorch users without prior experience of using Pyro.

In this section we provide more details on each of the modelling steps along the example of a synthetic one-dimensional non-linear regression dataset. We use the setup from (Foong et al., 2019) with two clusters of inputs $x_1 \sim \mathcal{U}[-1, -0.7]$, $x_2 \sim \mathcal{U}[0.5, 1]$ and $y \sim \mathcal{N}(\cos(4x + 0.8), 0.1^2)$.

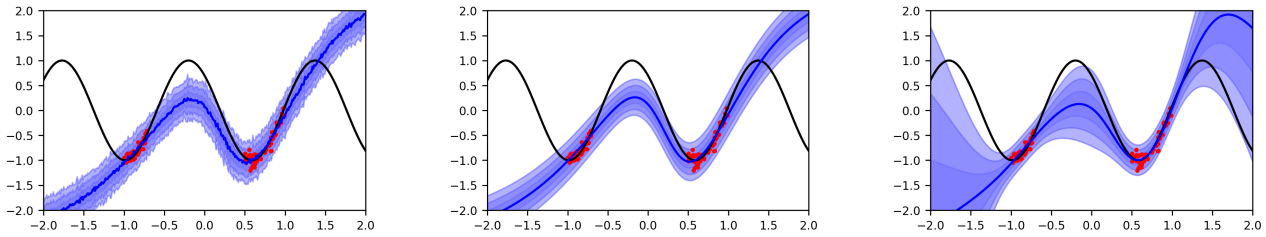
2.1 Defining a BNN

A TyXe BNN has four components: a PyTorch neural network, a data likelihood, a weight prior and a guide[1] factory for the posterior. We describe their signature and our instantiations below. As seen in Listing 1, turning a PyTorch network into a TyXe BNN requires as little as five lines of code.

2.1.1 Network architecture

PyTorch provides a range of classes that facilitate the construction of neural networks, ranging from simple linear or convolutional layers and nonlinearities as building blocks to higher-level classes that compose these, e.g. by chaining them together in the `nn.Sequential` module. A simple regression network on $1d$ data with one layer of 50 hidden units and a `tanh` nonlinearity, as commonly used for illustration in works on Bayesian neural networks, can be defined in a single line of code (first line of Listing 1). More generally, any neural network in PyTorch is described by the `nn.Module` class, which provides functionalities such as easy composition, parameter and gradient handling, and many more conveniences for neural network researchers and practitioners that have contributed to the wide adoption of this framework. Further, the `torchvision` package implements various modern architectures, such as ResNets (He et al., 2016). TyXe can also work on top of architectures

[1] Following Pyro's terminology we refer to programs drawing approximate posterior samples "guides".



(a) Local reparameterization (b) Shared weight samples (c) HMC

Figure 1: Bayesian nonlinear regression using the setup from Listing 1 and fit using Listing 2. Fig. 1(a) wraps the call to `bnn.predict` in the local reparameterization context with the call to `fit`, Fig. 1(b) does not. Switching between the two is as simple as adapting the indentation of the call to `predict` to be in- or outside the `local_reparameterization` context. Both use the same bnn object with the same approximate posterior. Fig. 1(c) uses `pyro.infer.mcmc.HMC` as guide factory. The shaded area indicates up to three standard deviations from the predictive mean.

from 3rd party libraries, such as DGL (Wang et al., 2019), that derive from `nn.Module`.

Pyro inherits the elegant abstractions for neural networks from PyTorch through its `PyroModule` class, which extends `nn.Module` to allow for instance attributes to be modified by Pyro effect handlers, making it easy to replace `nn.Parameter`s with Pyro sample sites. We adopt the `PyroModule` class under the hood to provide a seamless interface between TyXe and PyTorch networks.

2.1.2 *Prior*

At this time, we restrict the probabilistic model definition to weight space priors. Our classes take care of constructing distribution objects that replace the network parameters as `PyroSample`s. One such prior class is an `IIDPrior` which takes a Pyro distribution as argument, such as a `pyro.distributions.Normal(0., 1.)`, applying a standard normal prior over all network parameters. We further implement `LayerwiseNormalPrior`, a per-layer Gaussian prior that sets the variance to the inverse of the number of input units as recommended in (Neal, 1996), or analogous to the variance used for weight initialization in (Glorot & Bengio, 2010; He et al., 2015) when using the flag `method={"radford", "xavier", "kaiming"}`, respectively. Crucially, we do not require users to set priors for each layer by hand, this is dealt with automatically by our framework.

Our prior classes accept arguments that allow for certain layers or parameters to be excluded from a Bayesian treatment. The prior in our ResNet example in Section 3 receives `hide_module_types=[nn.BatchNorm2d]` to hide the parameters of the BatchNorm modules. Those parameters stay deterministic and are fit to minimize the log likelihood part of the ELBO.

2.1.3 *Guide*

The guide argument is the only place where the initialization of our `VariationalBNN` and `MCMC_BNN` differs. `tyxe.VariationalBNN` expects a function that automatically constructs a guide for the network weights, e.g. a `pyro.infer.autoguide`, and an optional second such function for variables in the likelihood if present (e.g. an unknown variance in a Gaussian likelihood).

To facilitate local reparameterization and computation of KL-divergences in closed form, we implement an `AutoNormal` guide, which samples all unobserved sites in the model from a diagonal Normal. This is similar to Pyro's `AutoNormal` autoguide, which constructs an auxiliary joint latent variable with a factorized Gaussian distribution. Variational parameters can be initialized as for autoguides by sampling from the prior/estimating statistics like the prior median, or through additional convenience functions that we provide, such as sampling the means from distributions with variances depending on the numbers of units in the corresponding layers, akin to how deterministic layers are typically initialized. This also permits initializing means to the values of pre-trained networks, which is particularly convenient when converting a deep network into a BNN.

The `tyxe.MCMC_BNN` class expects an MCMC kernel as guide, either HMC (Neal, 2012) or NUTS (Hoffman & Gelman, 2014), and runs Pyro's MCMC on the full dataset to obtain samples from the posterior. For both `BNN` classes, arguments to the guide constructor can be passed via `partial` from Python's built-in `functools` module. Listing 3 shows an example of this.

2.1.4 *Likelihood*

Our likelihoods are wrappers around Pyro's `distributions`, expecting a `dataset_size` argument to correctly scale the KL term when using mini-batches. Specifically we provide Bernoulli, Categori-

```
optim = pyro.optim.Adam({"lr": lr})
with tyxe.poutine.\
    local_reparameterization():
  bnn.fit(loader, n_epochs, optim)
pred_params = bnn.predict(
  test_data, num_predictions=n)
```

Listing 2: Regression fit and predict example with local reparameterization enabled for training only.

cal, HomoskedasticGaussian and HeterosketdasticGaussian likelihoods. Implementing a new likelihood requires a `predictive_distribution(predictions)` method returning a Pyro distribution for sampling. Further, it should provide a method for calculating an error estimate for evaluation, such as the squared error for Gaussian models or classification error for discrete models. Hence it is easy to add new likelihoods based on existing distributions, e.g. a Poisson likelihood.

2.2 Fitting a BNN

Our BNN class provides a scikit-learn-style `fit` function to run inference for a given numbers of passes over an `Iterable`, e.g. a PyTorch `DataLoader`. Each element is a length-two tuple, where the first element contains the network inputs (and may be a list) and the second is the likelihood targets, e.g. class labels. The `VariationalBNN` class further requires a Pyro optimizer.

`tyxe.VariationalBNN` runs stochastic variational inference (Ranganath et al., 2014; Wingate & Weber, 2013), a popular training algorithm for Bayesian Neural Networks, e.g. (Blundell et al., 2015) based on maximizing the evidence lower bound (ELBO). Our implementation automatically handles correctly scaling the KL-term vs. the log likelihood in the ELBO. `tyxe.MCMC_BNN` provides a compatible interface to Pyro's `MCMC` class.

Listing 2 shows a call to `fit`. Besides the data loader and number of epochs or samples, it is possible to pass in a `callback` function to the `VariationalBNN`, which is invoked after every epoch with the average value of the ELBO over the epoch and can be used e.g. to check the log likelihood of a validation data set. By returning `True`, the callback function can stop training. The `MCMC_BNN` passes any keyword arguments on to Pyro's `MCMC` class.

2.3 Predicting with a BNN

The `predict` method returns predictions for a given number of weight samples from the approximate posterior. Listing 2 invokes `predict` at the bottom. By default it aggregates the sampled predictions, i.e. averages them. Via `aggregate=False` the sampled predictions can be returned in a stacked tensor. We further implement an `evaluate` method that expects test labels and returns their log likelihood along with an error measure depending on the model, e.g. squared error for Gaussian likelihoods and classification error for Categorical or Binary ones.

2.4 Transformations via effect handlers

One crucial component missing from Pyro that TyXe provides is BNN-specific effect handlers (Plotkin & Pretnar, 2009; Bingham et al., 2019), specifically local reparameterization (Kingma et al., 2015) and flipout (Wen et al., 2018) for gradient variance reduction. Local reparameterization samples the pre-activations of each data point rather than a single weight matrix shared across a mini-batch for factorized Gaussian approximate posteriors over the weights and layers performing linear mappings, such as dense or convolutional layers. Flipout, on the other hand, samples a rank-one matrix of signs per data point, which allows for using distinct weights in a computationally efficient manner in linear operations, if the weights are sampled from a factorized symmetric distribution.

Typically, these are implemented as separate layer classes, e.g. (Tran et al., 2019). This creates an unnecessary redundancy in the code base, since there are now two versions of the same model differing only in sampling approaches for gradient estimation at each linear mapping. From a probabilistic modeling point of view it is preferable to separate model and inference explicitly to facilitate reuse of models and inference approaches. Fortunately, Pyro provides an expressive module for effect handling, which we can leverage to modify the computation as required. Specifically, we implement a `LocalReparameterizationMessenger` which marks linear functions called by PyTorch modules, such as `F.linear`, as effectful in order to modify how linear computations are performed as required. The Messenger maintains references from samples to their distributions and, when a linear function is called in a `local_reparameterization` context on weights from a factorized Gaussian, samples the output from the Gaussian over the result of the linear mapping.

Listing 2 calls `fit` in such a context. The call to `predict` could be wrapped too, but the purpose of local reparameterization and flipout is to reduce gradient variance. As they double the computational cost, we omit them for testing.

3 Large-scale vision classification

The biggest advantage resulting from our choice not to implement bespoke layer classes is that implementations of popular architectures can immediately be turned into their Bayesian counterparts. While implementing the two-layer network from the regression example with Bayesian layers

```
1 resnet = torchvision.models.resnet18(pretrained=True)
2 prior = tyxe.priors.IIDPrior(dist.Normal(0, 1), expose_all=False,
3                              hide_module_types=[nn.BatchNorm2d])
4 likelihood = tyxe.likelihoods.Categorical(dataset_size)
5 guide = partial(tyxe.guides.AutoNormal, train_loc=False, init_scale=1e-4,
6   init_loc_fn=tyxe.guides.PretrainedInitializer.from_net(resnet))
7 bayesian_resnet = tyxe.VariationalBNN(resnet, prior, likelihood, guide)
8 ...  # alternative last-layer prior and guide below
9 ll_prior = tyxe.priors.IIDPrior(dist.Normal(0, 1), expose_all=False,
10   expose_modules=[resnet.fc])
11 lr_guide = pyro.infer.autoguide.AutoLowRankMultivariateNormal
```

Listing 3: Bayesian ResNet. Line 1 loads a ResNet with pre-trained parameters from `torchvision`. The prior in lines 2−3 excludes BatchNorm layers, keeping their parameters deterministic. Arguments to the guide are passed with `partial` as in lines 5−6. We show how to set the Gaussian means to the pre-trained weights and only fit the variances, which are initialized to be small. The `BNN` object in line 7 is constructed exactly the same way as in the regression example. Lines 9−11 show an alternative prior that only applies to the final fully-connected layer alongside a Pyro autoguide.

is of course not complicated, writing the code for a modern computer vision architecture, e.g. a ResNet (He et al., 2016), is significantly more cumbersome and error-prone. With TyXe, users can use the ResNet implementation available through `torchvision` as shown in Listing 3. In this example we further highlight the flexibility of TyXe to only perform inference over some parameters while keeping others deterministic by excluding `nn.BatchNorm2d` layers from a Bayesian treatment.

To showcase how the clean separation of network architecture, prior, guide and likelihood in TyXe facilitates an experimental workflow, we investigate the predictive uncertainty of different inference strategies for a Bayesian ResNet. In Listing 3 we define a fully factorized Gaussian guide that fixes the means to the values of pre-trained weights and only fits the variances as parameters. While we would usually want the approximate posterior to be as flexible as possible, it has been observed in the literature (Louizos & Welling, 2017; Trippe & Turner, 2018) that such restrictions can improve the predictive performance of a BNN. We further investigate a mean-field guide where we similarly initialize the means to pre-trained weight values, but do not fix them for optimization, and restrict the variance of the variational distribution to a maximum of 0.1 to prevent underfitting. Finally we test performing inference in only the final classification layer with a Gaussian guide with either a diagonal or low-rank plus diagonal covariance matrix (also shown in the Listing) while using the pre-trained weights for the previous layers. Switching between these options is easy, with typically only a single or two lines of code differing. As baselines we compare to maximum likelihood (ML) and maximum a-posteriori (MAP). For the full code see `examples/resnet.py`.

Fig. 2 compares calibration and entropy of the predictive distributions on test and out-of-distribution (OOD) data.

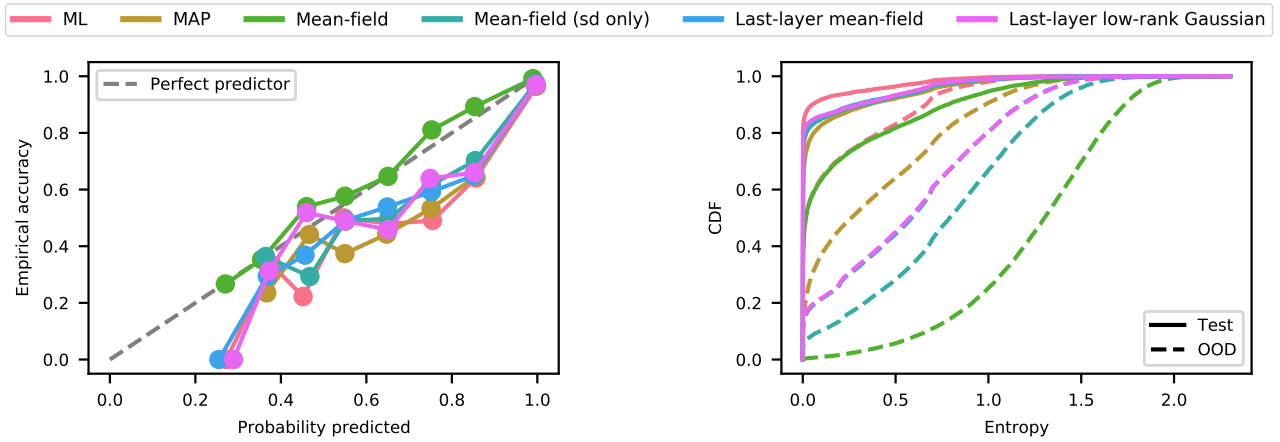
Table 1: Bayesian ResNet-18 predictive performance.

Inference	NLL↓	Acc.↑(%)	ECE↓(%)	OOD↑
ML	0.33	94.29	4.10	0.78
MAP	0.29	92.14	4.44	0.82
MF (sd only)	0.27	93.66	3.14	0.93
MF	0.20	93.28	0.97	0.94
LL MF	0.35	93.36	3.62	0.89
LL low rank	0.34	93.31	3.75	0.89

Mean-field (MF) with learned means leads to better calibrated predictions than variants (re-)using point estimates. It best distinguishes test from OOD data as measured by the area under the ROC curve based on the maximum predicted probability and has the lowest expected calibration error (ECE) and negative log likelihood (NLL), see Tab. 1.

We use the usual data augmentation techniques for CIFAR10 of randomly flipping and cropping the images after padding them with 4 pixels on each dimension and we normalize all channels to have zero-mean and unit-standard deviation. All methods use the Adam optimizer (Kingma & Ba, 2015). We train the deterministic inference methods (ML, MAP) for 200 epochs with a learning rate of 10^{-3} and another 100 epochs with a learning rate of 10^{-4}. All variational methods are trained for 200 epochs with a learning rate of 10^{-3} and we initialize the means to pre-trained ML parameters. We use a rank of 10 for the low-rank plus diagonal posterior and average over 32 samples for predictions on the test and OOD sets. The factorized Gaussian posteriors all use local reparameterization and we limit the standard deviation of the mean-field posteriors to 0.1.

We provide a pure Pyro snippet for a variational ResNet in Appendix B for a direct comparison. The implementation requires knowledge of a range of Pyro constructs to avoid pitfalls such as incorrectly scaling prior and likeli-



(a) Test calibration.

(b) Test vs. OOD uncertainty.

Figure 2: Calibration curves and empirical cumulative density of the entropy of the predictive distribution on test and OOD data for Bayesian ResNet-18 with different inference approaches on CIFAR10 (OOD: SVHN).

hood, yet the code ends up being significantly lengthier and somewhat convoluted. In contrast, TyXe provides a clean object-oriented interface that will be intuitive for most users with a basic understanding of Bayesian statistics and accessible for pure PyTorch users who do not want to have to learn Pyro. Crucially, essential features for achieving good discriminative performance with a BNN, such as reparameterization and clipping the variance of the approximate posterior are not available in Pyro.

4 Compatibility with external libraries

TyXe is compatible with libraries outside of the native PyTorch ecosystem and classical settings such as classification of i.i.d. images or regression, as long as the networks build on top of `nn.Module`. Below, we demonstrate this on a semi-supervised node classification example with a graph neural network from the DGL (Wang et al., 2019) tutorials, as well as a 3D rendering example in PyTorch3D.

4.1 Bayesian graph neural networks with DGL

We extend an example from the DGL tutorials[2] to train a Bayesian graph neural network (GNN) on the Cora dataset. Graph datasets are often semi-supervised, where an entire graph of nodes is provided, but only some of them are labelled. Hence we need a mechanism for preventing unlabelled nodes from contributing to the log likelihood. We combine Pyro's `block` and `mask` poutines to implement the `selective_mask` effect handler, which can wrap the call to `fit` as a context manager as shown in Listing 4 and mask out data in the likelihood. The network is taken from the DGL tutorial without change. As it utilizes `nn.Linear`, it is compatible with flipout. Prior, guide, likelihood and BNN can be constructed exactly as in the previous examples, see `examples/gnn.py` for the code.

In Tab. 2 we report NLLs, accuracies and ECE for ML, MAP and MF. ML leads to overfitting and requires the use of early stopping. Further it suffers from overconfident predictions, which can be mitigated to a degree by the use of variational inference, although not to the same extent as in the image classification example. Bayesian GNNs have only recently been started to be investigated in a few works (Zhang et al., 2019; Hasanzadeh et al., 2020; Luo et al., 2020; Lamb & Paige, 2020) and we believe that TyXe can be a valuable tool for putting Bayesian inference at the disposal of the graph neural network community.

Following the DGL tutorial, we train ML (and MAP) for 200 iterations with a learning rate of 10^{-2} using Adam. We report the test accuracy at the iteration with lowest validation negative log likelihood. For mean-field, we train for 400 iterations with an initial learning rate of 0.1, which we decay by a factor of 10 every 100 iterations and we limit the variational standard deviations to 0.3. Means are initialized to the random initialization of the deterministic network and we draw 8 posterior samples for evaluation. We use 10 bins to calculate the expected calibration error.

4.2 Custom losses: Bayesian NeRF with PyTorch3D

Next, we adapt a more complex example on Neural Radiance Fields (NeRF) (Mildenhall et al., 2020) from the

[2] https://docs.dgl.ai/en/0.5.x/tutorials/models/1_gnn/1_gcn.html

```python
class GCNLayer(nn.Module):
  ...
  def forward(self, graph, x):
    with graph.local_scope():
      graph.ndata['h'] = x
      graph.update_all(
        gcn_msg, gcn_reduce)
      h = graph.ndata['h']
      return self.linear(h)
```

```python
class GNN(nn.Module):
  ...
  def forward(self, graph, x):
    x = self.gcn_layer1(graph, x)
    x = torch.relu(x)
    return self.gcn_layer2(graph, x)
```

```
1 bgnn = tyxe.VariationalBNN(gnn, prior, guide, likelihood)
2 ...
3 with tyxe.poutine.selective_mask(mask=mask, expose=["likelihood.data"]):
4   bgnn.fit([((graph, x), y)], optim, n_epochs)
```

Listing 4: GNN example. The graph convolutional layer definition (top left) relies on DGL's graph functionality and is used for the GNN (top right). The Bayesian GNN can be constructed in line 1 with the exact same prior, guide and likelihood options as previously. The `selective_mask` in line 3 ensures that only predictions on labelled nodes contribute to the log likelihood when calling `fit` in line 4. The input data now consists of a graph and node features.

```
1 nerf_bnn = tyxe.PytorchBNN(nerf_net, prior, guide)
2 optim = torch.optim.Adam(nerf_bnn.pytorch_parameters(dummy_data), lr=1e-3)
3 ...
4 images, rays = renderer(cameras_batch, nerf_bnn)
5 image_loss = calc_loss(images, rays, targets)
6 loss = image_loss + scale * nerf_bnn.cached_kl_loss
7 loss.backward(); optim.step()
```

Listing 5: Bayesian NeRF example. Constructing a `PytorchBNN` is similar to a `VariationalBNN` in line 1 but without the likelihood. No downstream changes except for parameter collection for the PyTorch optimizer in line 2 – which requires a batch of data to trace parameters on a call to the net's forward method – are needed. The `nerf_bnn` can be passed into the PyTorch3D renderer in line 4 as a drop-in replacement for the `nerf_net`. The loss can be calculated as before in lines 5, with the possible addition of the KL regularizer in line 6. Automatic differentiation and parameter updates can be performed as in standard PyTorch code in line 7.

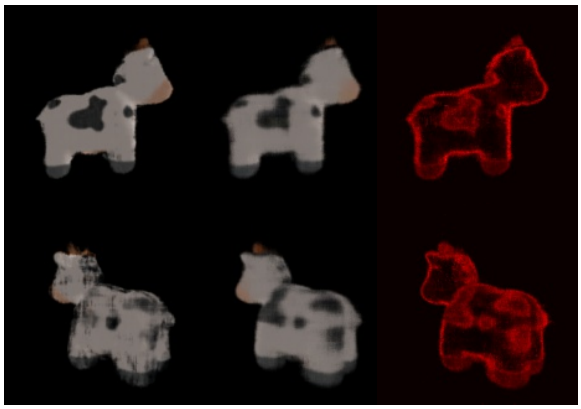
Table 2: Performance of deterministic and Bayesian GNNs on the Cora dataset. We report the lowest validation NLL along with the test accuracy and ECE at the corresponding epoch (mean and two standard errors over five runs).

Inference	NLL↓	Acc.↑	ECE↓
ML	$1.01 \pm .04$	75.64 ± 1.28	15.38 ± 0.97
MAP	$0.93 \pm .03$	75.94 ± 0.73	12.78 ± 0.96
MF	$0.77 \pm .02$	78.02 ± 1.00	10.22 ± 1.31

PyTorch3D repository[3] to train a Bayesian NeRF. The loss function does not straight-forwardly correspond to a probabilistic likelihood and is calculated as a custom error function of rendered image and silhouette. Hence there is no suitable likelihood class to implement for TyXe and it is not clear how the the prior or KL term should be weighed relative to the error. Therefore this example is not Bayesian in the proper sense as a 'posterior' as a product of likelihood and prior does not exist, but demonstrates that the uncertainty of a pseudo-Bayesian variational BNN can still improve the robustness on unseen data.

Specifically, we introduce a more low level `PytorchBNN` class that does not require a likelihood and can be used to directly wrap a PyTorch neural network. It is constructed similarly to `VariationalBNN` with a variational guide factory, but due to the absence of the likelihood does not provide convenience functions such as `fit` or `predict`. Instead, it is intended to serve as a drop-in replacement of the deterministic neural network in a PyTorch-based workflow. The output of the `forward` method corresponds to predictions of the network made with a single Monte Carlo sample from the variational posterior. The corresponding KL penalty term can be accessed through the `cached_kl_loss` attribute and added to the loss. It is

[3] https://github.com/facebookresearch/pytorch3d/blob/master/docs/tutorials/fit_simple_neural_radiance_field.ipynb



(a) Det. NeRF (b) Bay. NeRF (c) Uncertainty

Figure 3: PyTorch3D example. Top row seen during training, bottom row excluded. Bayesian NeRF achieves an error of 8.1×10^{-3} on a set of 10 held-out angles, while the error is 9.4×10^{-3} for the deterministic version. Uncertainty visualizes variance across different weight samples.

updated on every forward pass, i.e. when a sample is drawn from the approximate posterior. The key difference to a regular PyTorch neural network is that since Pyro initializes parameters lazily, we cannot provide a `parameters` method. Instead, optimizable parameters are collected via `pytorch_parameters`, which takes a batch of data to pass through the network for tracing the parameters.

We provide a code snippet in Listing 5. We emphasize that parameters are trained with the original PyTorch instead of a Pyro optimizer, further reducing the required changes to the original workflow. The renderer is a PyTorch3D object and uses the Bayesian NeRF object instead of the original PyTorch network. The data-dependent loss is then calculated as before and the KL-divergence of the approximate posterior from the prior on the weights can be added to the objective as a regularizer, possible weighed by some scalar `scale`. The full code can be found in `examples/nerf.py` and is identical to the original notebook for the most part, with only a few lines needing to be modified to adapt it to TyXe, as well as some additional plotting code for visualizing the predictive uncertainty.

In the original example, the network is trained to render views of a cow from 360°. We hold out 90° as out-of-distribution data. As Fig. 3 shows, this leads to many artifacts and discontinuities with a deterministic net. The pseudo-Bayesian NeRF averages many of these out, and provides helpful measures of uncertainty in form of the variances of the predicted images (right column).

We train the deterministic NeRF with the recommended settings from the tutorial, i.e. $20,000$ iterations with an initial learning rate of 10^{-3}, which is decayed by a factor of 10 for the final 5000 iterations. The Bayesian NeRF uses the same learning rate schedule. Means are initialized to the parameters of the deterministic NeRF and standard deviations to 10^{-2}. We linearly anneal the weight of the KL term over the first $10,000$ iterations to the inverse of the number of RGB values in the colour images plus the number silhouette pixels. We use 8 samples for test predictions and calculate averages and standard deviation over the final image outputs of the renderer.

5 Variational continual learning

Finally, we show how our separation of prior, guide and network architecture enables an elegant implementation of variational continual learning (VCL) (Nguyen et al., 2018). VCL performs online updates of a single variational posterior over a sequence of datasets by setting the prior to be equal to the posterior after training on a task. Hence, having set up and trained a BNN on a first task as in the previous examples, we only need construct a new prior from the guide distributions over the weights to update the previous BNN prior. We show example code for this process in Listing 6 and the full implementation can be found in `examples/vcl.py`. Training on the following task can then be conducted as usual with the `fit` method on the current dataset.

In Fig. 4 we show the test accuracy across the observed tasks after training on each one on the classical Split-MNIST and Split-CIFAR benchmarks(Zenke et al., 2017). We do not use coresets as (Nguyen et al., 2018), but this would only require some boilerplate code for creating the coresets prior to training and then fine-tuning on each coreset prior to testing by calling `fit` and restoring the state of the Pyro parameter store. As previously reported in the literature, deterministic networks suffer from forgetting on previous tasks, which can be mitigated by using a Bayesian approach such as VCL.

Following the recommendations in (Swaroop et al., 2019), we train on each MNIST task for 600 epochs and each CIFAR task for 60. We use Adam with a learning rate of 10^{-3}. The architecture on MNIST is a fully connected network with a hidden layer of 200 units with ReLU nonlinearities. The convolutional architecture on CIFAR has two blocks of $Conv - ReLU - Conv - ReLU - Maxpool$ followed by a fully connected layer with 512 units. The convolution layers in the first block have 32, in the second block 64 channels and all use 3×3 kernels with a stride and padding of 1. The maxpool operation is 2×2 with a stride of 2. We normalize all CIFAR tasks to have zero-mean and unit-standard deviation per channel and do not use any form of data augmentation.

```
1 bayesian_weights = tyxe.util.pyro_sample_sites(bnn.net)
2 posteriors = bnn.net_guide.get_detached_distributions(bayesian_weights)
3 bnn.update_prior(tyxe.priors.DictPrior(posteriors))
```

Listing 6: Updating the prior of a `BNN` for variational continual learning. Line 1 collects all weights over which we perform inference, line 2 extracts the corresponding variational distributions from the guide, and line 3 uses these to update the BNN's prior.

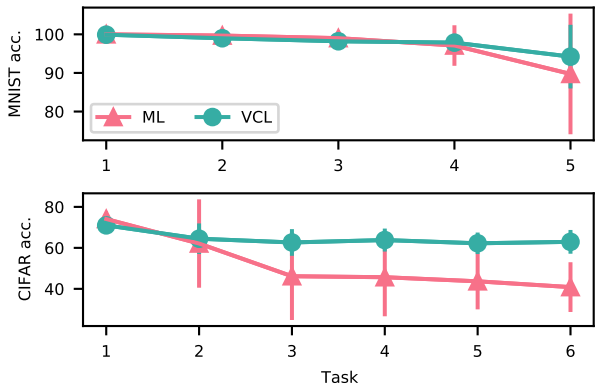


Figure 4: Mean accuracy and two standard errors on tasks seen so far for VCL and ML on Split-MNIST and -CIFAR.

6 Related work

The most closely related piece of recent work is Bayesian Layers (Tran et al., 2019), which extends the layer classes of Keras with the aim of them being usable as drop-in replacements for their deterministic counterpart. This forces the user to modify the code where the network is defined or write their own boilerplate code. Bayesian Layers are currently more general in scope, providing an abstraction over uncertainty over composable functions including normalizing flows and Gaussian Process mappings per layer, while at this point we have consciously limited ourselves to weight space uncertainty in neural networks and treat networks holistically rather than per layer.

For PyTorch, PyVarInf[4] provides functionality for turning `nn.Modules` into BNNs in a similar spirit to TyXe. As it is not backed by a probabilistic programming framework, the choice of prior distributions is limited, inference is restricted to variational factorized Gaussians, sampling tricks such as local reparameterization are not implemented and MCMC-based inference is not available. BLiTZ[5] (Esposito, 2020) provides variational counterparts to PyTorch's linear, convolutional and some recurrent layers. Networks need to be constructed manually based on those, with no support of other layer types. Priors are limited to mixtures of up to two Gaussians and inference is performed with a factorized Gaussian without support for gradient variance reduction techniques. More recently, LaplaceRedux (Daxberger et al., 2021a) provides support for modern variants (Ritter et al., 2018a;b; Daxberger et al., 2021b) of the Laplace approximation for neural networks (MacKay, 1992). Similarly to TyXe, the package builds on top of existing PyTorch networks, however inference is limited to the Laplace approximation and priors to Gaussian distributions. See Tab. 3 for a tabular comparison.

Subsequent to the initial code release of TyXe, UQ360 (Ghosh et al., 2021) and BNNPriors (Fortuin et al., 2021) were released. BNNPriors is a library focused on exploration of BNN priors and provides support for a range of different weight priors, restricting inference to (stochastic) MCMC-based methods; UQ360 provides a general armamentarium of uncertainty estimation techniques, including frequentist and non-parametric methods. Neither of them are focused on the PyTorch deep learning practitioner wanting to keep their workflow intact.

7 Future directions

In the long-term we view TyXe as a high-level BNN interface that complements Pyro with features specific to Bayesian deep learning that are not of interest for more general probabilistic programs. Below we discuss both some specific components as well as broader directions in which we plan to move TyXe in the future.

There is a wide range of BNN-specific variational inference approaches that has been developed in the literature over recent years. Some of these, e.g. (Swiatkowski et al., 2020) and (Tomczak et al., 2020) lend themselves particularly well to the abstractions that we have built and we plan to implement these methods. Especially the latter can be conveniently factorized into a general-purpose AutoGuide and logic that extends our local reparameterization effect handlers.

On a related note, we are interested in adding a layer of abstraction to Pyro's AutoGuides that simplifies the construction of AutoGuide classes with some family of distribution that is shared across all variables for inference. This feature could conveniently provide matrix normal posteriors (Louizos & Welling, 2016).

While Pyro provides some full-batch MCMC samplers such

[4] https://github.com/ctallec/pyvarinf

[5] https://github.com/piEsposito/blitz-bayesian-deep-learning

Table 3: Comparison of design features with related packages. * TyXe provides basic support for Laplace through Pyro's `AutoLaplaceApproximation`, which is however not scalable to larger networks.

	Bayesian layers	PyVarInf	BliTZ	BNNPriors	LaplaceRedux	TyXe
Supports existing nets	×	✓	×	×	✓	✓
Layer agnostic	×	×	×	✓	×	✓
Flexible priors	✓	×	×	✓	×	✓
Inference	VI	VI	VI	MCMC	Laplace	VI, MCMC, Laplace*

as HMC and NUTS, more scalable mini-batch methods are not available, such as SGLD (Welling & Teh, 2011). We intend to add the necessary abstractions and make them available through TyXe or directly contribute them to Pyro.

Our layer-free implementation of local reparameterization further offers the possibility of formulating and training stochastic binary neural networks (Shayer et al., 2018) in a Bayesian framework, which is an exiting direction of research that we intend to explore.

A pragmatic approach for uncertainty estimation in practice is Monte Carlo Dropout (Gal & Ghahramani, 2016). Typical implementations such as in PyTorch implicitly draw a single weight sample per input, however for visualization purposes it can be desirable to fix a single sample across batches of data. Registering Dropout layers as an effect handler could give access to this functionality through Pyro's poutine library.

To further enhance TyXe's scope and support research on BNNs in addition to applying them to practical problems, we are highly interested in exploring if moment propagation approaches such as (Hernández-Lobato & Adams, 2015) and (Wu et al., 2019) can be implemented as effect handlers in a similar spirit to our reparameterization poutines. This may require marking nonlinearity functions as effectful in addition to linear and convolutional layers in order to allow passing distributions rather than tensors through a given network and could greatly facilitate experimenting with such approaches on a broad range architectures with existing PyTorch implementations.

8 Conclusion

We have presented TyXe, a Pyro-based library that facilitates a seamless integration of Bayesian neural networks for uncertainty estimation and continual learning into PyTorch-based workflows. We have demonstrated the flexibility of TyXe with applications based on 3rd-party libraries, ranging from modern deep image classification architectures over graph neural networks to neural radiance fields, an important model class in 3d-vision. In all cases the workflow of a practitioner knowledgeable in the use of these libraries is minimally impacted to incorporate TyXe and transform their systems to incorporate Bayesian neural networks. TyXe avoids implementing bespoke layer classes and instead leverages and expands on Pyro's powerful effect handler module, resulting in a flexible design that cleanly separates architecture definition, prior, inference, likelihood and sampling logic.

TyXe's choices of variational distributions are currently pragmatic, focused on serving practitioners and researchers interested in generating uncertainty estimates for downstream tasks that will benefit from the improvements offered by standard variational families or HMC over maximum likelihood. Recent work has even argued that mean-field may be sufficient for inference in deep networks (Farquhar et al., 2020). However, we are highly interested in further developing TyXe to support more complex recent approaches and become a tool for Bayesian deep learning research with its backing by Pyro facilitating extensions as discussed in-depth Section 7. We would expect techniques with structured covariance matrices (Louizos & Welling, 2016; Ritter et al., 2018b) as well as hierarchical weight models (Louizos & Welling, 2017; Karaletsos et al., 2018; Ritter et al., 2021) to be feasible to express within TyXe, with the latter possibly requiring additional abstractions. Nevertheless, we believe that similar to Bayesian Layers (Tran et al., 2019) TyXe already makes a valuable contribution to the ML software ecosystem, filling the gap of easy-to-use uncertainty estimation for PyTorch.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. Pyro: Deep universal probabilistic programming. *JMLR*, 2019.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. Weight uncertainty in neural network. In *ICML*, 2015.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J.,

Li, P., and Riddell, A. Stan: A probabilistic programming language. *Journal of Statistical Software*, 2017.

Chollet, F. et al. Keras, 2015.

Daxberger, E., Kristiadi, A., Immer, A., Eschenhagen, R., Bauer, M., and Hennig, P. Laplace Redux-effortless Bayesian deep learning. *NeurIPS*, 2021a.

Daxberger, E., Nalisnick, E., Allingham, J. U., Antorán, J., and Hernández-Lobato, J. M. Bayesian deep learning via subnetwork inference. In *ICML*, 2021b.

Esposito, P. BLiTZ - Bayesian layers in Torch zoo (a Bayesian deep learing library for Torch). https://github.com/piEsposito/blitz-bayesian-deep-learning/, 2020.

Farquhar, S., Smith, L., and Gal, Y. Liberty or depth: Deep Bayesian neural nets do not need complex weight posterior approximations. In *NeurIPS*, 2020.

Foong, A. Y., Li, Y., Hernández-Lobato, J. M., and Turner, R. E. 'In-between' uncertainty in Bayesian neural networks. *arXiv preprint arXiv:1906.11537*, 2019.

Fortuin, V., Garriga-Alonso, A., van der Wilk, M., and Aitchison, L. BNNpriors: A library for Bayesian neural network inference with different prior distributions. *Software Impacts*, 2021.

Gal, Y. and Ghahramani, Z. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *ICML*, 2016.

Ghosh, S., Liao, Q. V., Ramamurthy, K. N., Navratil, J., Sattigeri, P., Varshney, K. R., and Zhang, Y. Uncertainty quantification 360: A holistic toolkit for quantifying and communicating the uncertainty of AI, 2021.

Glorot, X. and Bengio, Y. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

Hasanzadeh, A., Hajiramezanali, E., Boluki, S., Zhou, M., Duffield, N., Narayanan, K., and Qian, X. Bayesian graph neural networks with adaptive connection sampling. In *ICML*, 2020.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *CVPR*, 2016.

Hernández-Lobato, J. M. and Adams, R. Probabilistic backpropagation for scalable learning of Bayesian neural networks. In *ICML*, 2015.

Hoffman, M. D. and Gelman, A. The No-U-Turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *JMLR*, 2014.

Karaletsos, T., Dayan, P., and Ghahramani, Z. Probabilistic meta-representations of neural networks. *arXiv preprint arXiv:1810.00555*, 2018.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. In *ICLR*, 2015.

Kingma, D. P., Salimans, T., and Welling, M. Variational dropout and the local reparameterization trick. In *NeurIPS*, 2015.

Lamb, G. and Paige, B. Bayesian graph neural networks for molecular property prediction. *arXiv preprint arXiv:2012.02089*, 2020.

Louizos, C. and Welling, M. Structured and efficient variational deep learning with matrix Gaussian posteriors. In *ICML*, 2016.

Louizos, C. and Welling, M. Multiplicative normalizing flows for variational Bayesian neural networks. In *ICML*, 2017.

Luo, Y., Huang, Z., Zhang, Z., Wang, Z., Baktashmotlagh, M., and Yang, Y. Learning from the past: Continual meta-learning with Bayesian graph neural networks. In *AAAI*, 2020.

MacKay, D. J. A practical Bayesian framework for backpropagation networks. *Neural Computation*, 1992.

Mildenhall, B., Srinivasan, P. P., Tancik, M., Barron, J. T., Ramamoorthi, R., and Ng, R. NeRF: Representing scenes as neural radiance fields for view synthesis. In *ECCV*, 2020.

Neal, R. M. Priors for infinite networks. In *Bayesian Learning for Neural Networks*. 1996.

Neal, R. M. *Bayesian learning for neural networks*. 2012.

Nguyen, C. V., Li, Y., Bui, T. D., and Turner, R. E. Variational continual learning. In *ICLR*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. In *NeurIPS*, 2019.

Plotkin, G. and Pretnar, M. Handlers of algebraic effects. In *European Symposium on Programming*, 2009.

Ranganath, R., Gerrish, S., and Blei, D. Black box variational inference. In *AISTATS*, 2014.

Ritter, H., Botev, A., and Barber, D. Online structured laplace approximations for overcoming catastrophic forgetting. *NeurIPS*, 2018a.

Ritter, H., Botev, A., and Barber, D. A scalable Laplace approximation for neural networks. In *ICLR*, 2018b.

Ritter, H., Kukla, M., Zhang, C., and Li, Y. Sparse uncertainty representation in deep learning with inducing weights. *arXiv preprint arXiv:2105.14594*, 2021.

Shayer, O., Levi, D., and Fetaya, E. Learning discrete weights using the local reparameterization trick. In *ICLR*, 2018.

Swaroop, S., Nguyen, C. V., Bui, T. D., and Turner, R. E. Improving and understanding variational continual learning. *arXiv preprint arXiv:1905.02099*, 2019.

Swiatkowski, J., Roth, K., Veeling, B. S., Tran, L., Dillon, J. V., Mandt, S., Snoek, J., Salimans, T., Jenatton, R., and Nowozin, S. The k-tied normal distribution: A compact parameterization of Gaussian mean field posteriors in Bayesian neural networks. In *ICML*, 2020.

Tomczak, M. et al. Efficient low rank Gaussian variational inference for neural networks. In *NeurIPS*, 2020.

Tran, D., Hoffman, M. D., Moore, D., Suter, C., Vasudevan, S., Radul, A., Johnson, M., and Saurous, R. A. Simple, distributed, and accelerated probabilistic programming. In *NeurIPS*, 2018.

Tran, D., Dusenberry, M., van der Wilk, M., and Hafner, D. Bayesian layers: A module for neural network uncertainty. In *NeurIPS*, 2019.

Trippe, B. and Turner, R. Overpruning in variational Bayesian neural networks. *arXiv preprint arXiv:1801.06230*, 2018.

Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., and Zhang, Z. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *arXiv preprint arXiv:1909.01315*, 2019.

Welling, M. and Teh, Y. W. Bayesian learning via stochastic gradient Langevin dynamics. In *ICML*, 2011.

Wen, Y., Vicol, P., Ba, J., Tran, D., and Grosse, R. Flipout: Efficient pseudo-independent weight perturbations on mini-batches. In *ICLR*, 2018.

Wingate, D. and Weber, T. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

Wu, A., Nowozin, S., Meeds, E., Turner, R. E., Hernández-Lobato, J. M., and Gaunt, A. L. Deterministic variational inference for robust Bayesian neural networks. In *ICLR*, 2019.

Zenke, F., Poole, B., and Ganguli, S. Continual learning through synaptic intelligence. In *ICML*, 2017.

Zhang, Y., Pal, S., Coates, M., and Ustebay, D. Bayesian graph convolutional neural networks for semi-supervised classification. In *AAAI*, 2019.

A OUTLINE OF THE CURRENT CODEBASE

In this section we give an overview of the most relevant modules and classes in our codebase to guide the interested reader through our design and towards aspects of the library that are most relevant to them and discuss some implementation details.

`tyxe/bnn.py` This module contains our top-level BNN classes. They mostly act as wrappers leveraging the functionality of our `Prior` classes to turn given PyTorch `nn.Module`s into `PyroModule`s and define the probabilistic models to perform inference in. Further, they provide high-level functionality for training, prediction and evaluation, which is delegated to the `Likelihood` classes and Pyro.

- **_BNN** Base class for all TyXe BNNs. Turns a PyTorch neural network into a `PyroModule` given a `Prior` and gives access to a forward pass through the network with samples from the prior.
- **GuidedBNN** Base class on top of `_BNN` that gives access to a forward pass with the network given a trace (i.e. sample) from some inference procedure for the network.
- **PytorchBNN** Class for constructing objects that can act as variational drop-in replacements of PyTorch network objects in existing codebases. The forward method acts like the forward method of an `nn.Module` object, except that the weights and therefore the function itself are stochastic. Under the hood, as the BNN class has no control over whether the guide function returns the network output (for example, Pyro's AutoGuides do not) it makes the forward pass have the side-effect of caching the output with the object, so that it can be returned. Similarly it stores the KL divergence between approximate posterior and prior (which may be approximated stochastically in each forward pass as the difference of the log densities with the sample from the approximate posterior if the KL is not available in closed form) to use as a regularization term. A further implementation challenge is collecting parameters as in the `.pytorch_parameters` method, which is the equivalent of `.parameters` on an `nn.Module`. As Pyro programs, e.g. the AutoGuides, typically initialize their parameters, a sample batch of data is required to run both prior (which may have parameters to be estimated via maximum likelihood) and posterior and capture all parameters.
- **_SupervisedBNN** Base class inheriting from `GuidedBNN` that now also incorporates a `Likelihood` describing a model for observed data that is conditioned on the output of the neural network. Defines an API for a `predict` method that will run forward passes through the network for multiple posterior samples and either return them as a stacked tensor or aggregate them in a likelihood-dependent way (e.g. average them for predicted class probabilities).
- **VariationalBNN** Class for variational Bayesian neural networks in a supervised learning setting. Allows for an additional guide constructor to be passed in for the likelihood if it contains any variables to be inferred (e.g. the unknown variance of a Gaussian observation model). Further provides a `.fit` method that wraps Pyro's `SVI` and `Trace_ELBO` objects to minimize the variational lower bound w.r.t. parameters.
- **MCMC_BNN** Class for MCMC-based BNNs in a supervised learning setting based on Pyro's MCMC kernels. Similarly provides `fit` and `predict` methods that run MCMC on some data and make predictions using those samples respectively.

`tyxe/likelihoods.py` The likelihood classes are designed as high-level wrapper around `pyro.distributions` that take care of constructing a Pyro function for sampling in a forward, i.e. describing a probabilistic program for the data. Crucially by providing the dataset size, this handles correctly scaling the log likelihood against the log prior (or KL divergence in variational inference) for mini-batches of data. Further they implement logic for evaluating test predictions through the log likelihood and some error measure and aggregating multiple predictions.

- **Likelihood** Base class for all likelihood classes. Implements all high-level functionality around model construction and evaluation. Expects subclasses to provide functions that construct distribution objects from given network predictions as well handling aggregating multiple predictions and providing an error function.
- **_Discrete** Base class for Bernoulli and Categorical that handles all classification-related logic of error calculation and averaging of predicted probabilities or logits.
- **Bernoulli** Likelihood class for binary observations.
- **Categorical** Likelihood class for categorical observations.
- **Gaussian** Base class for Gaussians. Uses the squared error as the error measures and aggregates predictions to a mean and standard deviation.
- **HeteroskedasticGaussian** Gaussian likelihood that assume $2d$ dimensional predictions for d dimen-

sional observation, with the first half of the dimension encoding a mean and the second half the standard deviation. Uses the predicted standard deviations to weigh means according to their precision when aggregating.

HomoskedasticGaussian Gaussian likelihood that assumes a shared variance for all observations. Crucially, this variance may be a probabilistic function that places a prior on an unknown variance in order to support inference over this additional variable.

`tyxe/priors.py` The prior module provides classes that handle logic around replacing `nn.Parameter` attributes of `nn.Module` objects with `PyroSample` with some prior distribution when turning a network into a `PyroModule`. They further support updating the prior attributes of the `PyroSamples` in order to facilitate continual learning.

Prior Base class for prior classes that implements apply and update functions for the conversion and updating of `nn.Modules`. Further handles hide/expose functionality (following the logic of Pyro's `block` poutine) that allows excluding/including different network parameters based on their module instance (i.e. a specific layer), their module type (i.e. a specific layer class), their attribute name or their full name. This way entire layers classes can be excluded from a Bayesian treatment, e.g. BatchNorm layers, only specific layers can be considered, e.g. the final layer, or specific parameters can be left to be learned via maximum likelihood estimation, e.g. the bias terms and gives the user high flexibility for their probabilistic model specification through a simple and compact interface.

IIDPrior Class for i.i.d. prior across all parameters based on a given distribution object, e.g. `dist.Normal(0, 1)`.

LayerwiseNormalPrior Convenience class for per-layer i.i.d. Gaussian priors with variance depending on weight shape, e.g. inversely proportional to the number of input dimensions.

DictPrior Convenience wrapper around dictionaries to map parameter names to distributions, e.g. for continual learning.

LambdaPrior Convenience wrapper around functions that dynamically generate a distribution for a given parameter object.

`tyxe/guides.py` This module builds on top of Pyro's autoguide module. It provides an equivalent to the **AutoNormal** class that samples from the approximate posterior directly rather than transforming samples and wrapping them in a Delta distribution, order to be compatible with local reparameterization and calculating KL divergences in closed form. Further it provides convenience features such as limiting the variance of the learned posterior or only learning means or variances, which are not generally of interest for Bayesian inference as they reduce rather than increase the flexibility of the variational posterior, however these can improve the discriminative performance of Bayesian neural networks which may outweigh concerns over approximating the true posterior as closely as possible. Finally, the class provides a method for returning a dictionary mapping sampling site names to distributions with detached parameters, which simplifies turning a guide object into a prior. The module further provides some neural-network style initialization functions for variational mean parameters.

`tyxe/poutine/reparameterization_messenger.py` Effect handlers for reparameterization of certain linear operations. Specifically, these replace samples from weight distributions with samples from the distributions over the outputs, which reduces gradient variance. We implement these operations as 'Messenger' classes to be compatible with Pyro's poutine library.

_ReparameterizationMessenger Base class that marks PyTorch functions as effectful to register them with Pyro's effect handling stack and handles basic logic around catching reparameterizable sites (currently only fully-factorized Gaussian and Delta distributions). The core idea is to monkey-patch PyTorch functions used by linear layers such as `nn.Linear` and `nn.Conv` with a version of the corresponding `F.linear` and `F.conv` function wrapped in Pyro's `effectful` decorator. This does not modify any behaviour outside of Pyro functions, but allows for subclasses of `Messenger` to modify the behavious of these functions at runtime. We use this to maintain a mapping from samples of tensor to their respective sampling distributions to check if the weights of a reparameterizable function come from a compatible distribution. If that is the case, we delegate sampling the corresponding output to a `_reparameterize` method that is to be implemented in a subclass.

LocalReparameterizationMessenger Implements local reparameterization (Kingma et al., 2015) logic for reparameterization.

FlipoutMessenger Implements flipout (Wen et al., 2018) logic for reparameterization.

B COMPARISON TO PYRO

Here, we show an example of a variational ResNet in pure Pyro and discuss some of the differences to the TyXe version. We stress that Listing 3 and Listing 7 are **not** equivalent, as Pyro lacks an implementation of local reparemterization and its autoguide classes do not support clipping the variance of a variational guide, so custom implementations of these features would be required in order to achieve competitive discriminative performance.

While TyXe has an object-oriented interface for constructing prior, likelihood, guide and finally the BNN, with classes for the typical components of a Bayesian modelling workflow, Pyro requires the user to modify their neural network in-place and manually set prior distributions as attributes of the object itself. The likelihood then requires defining a function that calls the (now Bayesian) network in a probabilistic program to pass into Pyro's SVI class. Finally, TyXe removes the need for boilerplate code when training and making predictions. In particular the latter again requires either familiarity Pyro's poutine library or the Predictive class to perform a standard step of the Bayesian deep learning workflow (which is arguably less common in a smaller-scale Bayesian modelling workflow for which Pyro was primarily designed, where we may be interested in modelling and making inferences on a given dataset rather than predicting on held-out test data). Overall, we are convinced that TyXe makes Pyro's functionality significantly more accessible for PyTorch users interested in adding Bayesian methods to their workflow, in particular for those that have been primarily focused on pure deep learning, as TyXe has been designed with Bayesian deep learning at its core. Finally, TyXe fills critical gaps in the functionality Pyro for successfully training performant BNNs and further provides support for continual learning.

```
resnet = torchvision.models.resnet18(pretrained=True)
values = resnet.state_dict()
pyro.nn.module.to_pyro_module(resnet)

# prior definition
for m in resnet.modules():
    if isinstance(m, (nn.Linear, nn.Conv2d)):
        m.weight = pyro.nn.PyroSample(dist.Normal(
            torch.zeros_like(m.weight),
            torch.ones_like(m.weight)).to_event())
        m.bias = pyro.nn.PyroSample(dist.Normal(
            torch.zeros_like(m.bias),
            torch.ones_like(m.bias)).to_event())

# likelihood
def model(x, y=None):
    logits = resnet(x)
    with pyro.plate("data_plate", dataset_size):
        pyro.sample("data", dist.Categorical(logits=logits), obs=y)
    return logits

# guide
guide = pyro.infer.autoguide.AutoNormal(
        model, init_scale=1e-4,
        init_loc_fn=pyro.infer.autoguide.init_to_value(values=values))

# optimization
optim = pyro.optim.Adam({"lr": 1e-3})
svi = pyro.infer.SVI(model, guide, optim, pyro.infer.Trace_ELBO())

for _ in range(num_epochs):
    for x, y in iter(loader):
        svi.step(x, y)

# prediction
test_predictions = []
for _ in range(num_test_samples):
    tr = pyro.poutine.trace(guide).get_trace(x_test)
    test_predictions.append(pyro.poutine.replay(model, trace=tr)(x_test))
test_p = torch.stack(test_predictions).softmax(-1).mean(0)
```

Listing 7: Code snippet for defining and training a pure Pyro variational ResNet.