Aboleth Documentation

Release 0.7.0

Data61

Dec 20, 2017

Contents

A bare-bones TensorFlow framework for *Bayesian* deep learning and Gaussian process approximation[1] with stochastic gradient variational Bayes inference[2].

[1] Cutajar, K. Bonilla, E. Michiardi, P. Filippone, M. Random Feature Expansions for Deep Gaussian Processes. In ICML, 2017.
[2] Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. In ICLR, 2014.

CHAPTER 1

Features

Some of the features of Aboleth:

- Bayesian fully-connected, embedding and convolutional layers using SGVB[2] for inference.
- Random Fourier and arc-cosine features for approximate Gaussian processes. Optional variational optimisation of these feature weights as per[1].
- Imputation layers with parameters that are learned as part of a model.
- Very flexible construction of networks, e.g. multiple inputs, ResNets etc.
- Optional maximum-likelihood type II inference for model parameters such as weight priors/regularizers and regression observation noise.
- Compatible and interoperable with other neural net frameworks such as Keras (see the demos for more information).

CHAPTER 2

Why?

The purpose of Aboleth is to provide a set of high performance and light weight components for building Bayesian neural nets and approximate (deep) Gaussian process computational graphs. We aim for *minimal* abstraction over pure TensorFlow, so you can still assign parts of the computational graph to different hardware, use your own data feeds/queues, and manage your own sessions etc.

Here is an example of building a simple Bayesian neural net classifier with one hidden layer and Normal prior/posterior distributions on the network weights:

```
import tensorflow as tf
import aboleth as ab

# Define the network, ">>" implements function composition,
# the InputLayer gives a kwarg for this network, and
# allows us to specify the number of samples for stochastic
# gradient variational Bayes.
net = (
    ab.InputLayer(name="X", n_samples=5) >>
    ab.DenseVariational(output_dim=100) >>
    ab.Activation(tf.nn.relu) >>
    ab.DenseVariational(output_dim=1) >>
)

X_ = tf.placeholder(tf.float, shape=(None, D))
Y_ = tf.placeholder(tf.float, shape=(None, 1))

# Build the network, nn, and the parameter regularization, kl
nn, kl = net(X=X_)

# Define the likelihood model
likelihood = tf.distributions.Bernoulli(logits=nn)

# Build the final loss function to use with TensorFlow train
loss = ab.elbo(likelihood, Y_, N, kl)
```

```
# Now your TensorFlow training code here!
...
```

At the moment the focus of Aboleth is on supervised tasks, however this is subject to change in subsequent releases if there is interest in this capability.

CHAPTER 3

Installation

To get up and running quickly you can use pip and get the Aboleth package from PyPI:

```
$ pip install aboleth
```

For the best performance on your architecture, we recommend installing TensorFlow from sources.

Or, to install additional dependencies required by the demos:

```
$ pip install aboleth[demos]
```

To install in develop mode with packages required for development we recommend you clone the repository from GitHub:

```
$ git clone git@github.com:data61/aboleth.git
```

Then in the directory that you cloned into, issue the following:

```
$ pip install -e .[dev]
```

CHAPTER 4

Getting Started

See the quick start guide to get started, and for more in depth guide, have a look at our tutorials. Also see the demos folder for more examples of creating and training algorithms with Aboleth.

The full project documentation can be found on readthedocs.

CHAPTER 5

References

CHAPTER 6

License

Copyright 2017 CSIRO (Data61)

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

CHAPTER 7

Documentation Contents

7.1 Installation

Firstly, make sure you have TensorFlow installed, preferably compiled specifically for your architecture, see installing TensorFlow from sources.

To get up and running quickly you can use pip and get the Aboleth package from PyPI:

```
$ pip install aboleth
```

Or, to install additional dependencies required by the demos:

```
$ pip install aboleth[demos]
```

To install in develop mode with packages required for development we recommend you clone the repository from GitHub:

```
$ git clone git@github.com:data61/aboleth.git
```

Then in the directory that you cloned into, issue the following:

```
$ pip install -e .[dev]
```

Or:

```
$ pip install -e .[dev,demos]
```

If you also want to develop with the demos.

7.2 Quick Start Guide

In Aboleth we use function composition to compose machine learning models. These models are callable python classes that when called return a TensorFlow computational graph (really a `tf.Tensor`). We can best demonstrate

this with a few examples.

7.2.1 Logistic Classification

For our first example, lets make a simple logistic classifier with L_2 regularisation on the model weights:

```
import tensorflow as tf
import aboleth as ab

layers = (
    ab.InputLayer(name="X") >>
    ab.DenseMap(output_dim=1, l1_reg=0, l2_reg=.05) >>
    ab.Activation(tf.nn.sigmoid)
)
```

Here the right shift operator, >>, implements functions composition (or specifically, a writer monad) from the innermost function to the outermost. The above code block has has implemented the following function,

$$p(\mathbf{y} = 1|\mathbf{X}) = \sigma(\mathbf{X}\mathbf{w}),$$

where $\mathbf{w} \in \mathbb{R}^D$ are the model weights, $\mathbf{y} \in \mathbb{N}_2^N$ are the binary labels, $\mathbf{X} \in \mathbb{R}^{N \times D}$ are the predictive inputs and $\sigma(\cdot)$ is a logistic sigmoid function. At this stage `layers` is a callable class (`ab.baselayers.MultiLayerComposite`), and no computational graph has been built. `ab.InputLayer` allows us to name our inputs so we can refer to them later when we call our class `layers`. This is useful when we have multiple inputs into our model, for examples, if we want to deal with continuous and categorical features separately (see *Multiple Input Data*).

So now we have defined the structure of the predictive model, if we wish we can create its computational graph,

```
net, reg = layers(X=X_)
```

where the keyword argument `X` was defined in the `InputLayer` and `X_` is a placeholder (`tf.placeholder`) or the actual predictive data we want to build into our model. `net` is the resulting computational graph of our predictive model/network, and `reg` are the regularisation terms associated with the model parameters (layer weights in this case).

If we wanted, we could evaluate `net` right now in a TensorFlow session, however none of the weights have been fit to the data. In order to fit the weights, we need to define a loss function. For this we need to define a likelihood model for our classifier, here we choose a Bernoulli distribution for our binary classifier (which corresponds to a log-loss):

```
likelihood = tf.distributions.Bernoulli(probs=net)
```

If we were to call `likelihood.log_prob(Y)`, it would return a tensor that implements the log of a Bernoulli probability mass function,

$$\mathcal{L}(y_n, p_n) = y_n \log p_n + (1 - y_n) \log(1 - p_n).$$

which is an integral part of our loss function. Here we have used p_n as shorthand for $p(y_n = 1)$.

Note: We actually find it is more numerically stable to define Bernoulli likelihoods with logits:

```
likelihood = tf.distributions.Bernoulli(logits=net)
```

Where:

```
layers = (
    ab.InputLayer(name="X") >>
    ab.DenseMap(output_dim=1, l1_reg=0, l2_reg=.05) >>
```

```
)
net, reg = layers(X=X_)
```

The `Bernoulli` class then computes the sigmoid activation internally.

Now we have enough to build the loss function we will use to optimize the model weights:

```
loss = ab.max_posterior(likelihood, Y_, reg)
```

This is a *maximum a-posteriori* loss function, which can be thought of as a maximum likelihood objective with a penalty on the magnitude of the weights from a Gaussian prior (controlled by `l2_reg` or λ),

$$\min_{\mathbf{w}} -\frac{1}{N}\sum_{n}\mathcal{L}(y_n, \sigma(\mathbf{x}_n^\top \mathbf{w})) + \frac{\lambda}{2}\|\mathbf{w}\|_2^2.$$

Now we have enough to use the `tf.train` module to learn the weights of our model:

```
optimizer = tf.train.AdamOptimizer()
train = optimizer.minimize(loss)

with tf.Session() as sess:
    tf.global_variables_initializer().run()

    for _ in range(1000):
        sess.run(train, feed_dict={X_: X, Y_: Y})
```

This will run 1000 iterations of stochastic gradient optimization (using the Adam learning rate algorithm) where the model sees all of the data every iteration. We can also run this on mini-batches, see `ab.batch` for a simple batch generator, or TensorFlow's *train* and *data* modules for a more comprehensive set of utilities (we recommend looking at tf.train.MonitoredTrainingSession, and tf.data.Dataset)

Now that we have learned our classifier's weights, $\hat{\mathbf{w}}$, we will probably want to use for predicting class label probabilities on unseen data $\mathbf{x}^*$,

$$p(y^* = 1|\mathbf{X}, \mathbf{x}^*) = \sigma(\mathbf{x}^{*\top}\hat{\mathbf{w}}).$$

This can be very easily achieved by just evaluating our model on the unseen predictive data (still in the TensorFlow session from above):

```
probs = net.eval(feed_dict={X_: X_query})
```

However, you may find that `probs.shape` will be something like `(1, N, 1)` where `N = len(X_query)`. Aboleth made a new, 0th, axis here, and we'll talk about why this is the case in the next section.

Note: If you used logits as per the above note, then the prediction becomes:

```
probs = likelihood.probs.eval(feed_dict={X_: X_query})
```

And that is it!

7.2.2 Bayesian Logistic Classification

Aboleth is all about Bayesian inference, so now we'll demonstrate how to make a variational inference version of the logistic classifier. Now we explicitly place a prior distribution on the weights,

$$p(\mathbf{w}) = \mathcal{N}(\mathbf{w}|\mathbf{0}, \psi^2\mathbf{I}_D).$$

Here ψ is the prior weight standard deviation (note that this corresponds to $\sqrt{\lambda^{-1}}$ in the MAP logistic classifier). We use the same likelihood model as before,

$$p(y_n|\mathbf{w}, \mathbf{x}_n) = \text{Bernoulli}(y_n|\sigma(\mathbf{x}_n^\top\mathbf{w})),$$

and ideally we would like to infer the posterior distribution over these weights using Bayes' rule (as opposed to just the MAP value, $\hat{\mathbf{w}}$),

$$p(\mathbf{w}|\mathbf{X}, \mathbf{y}) = \frac{p(\mathbf{w})\prod_n p(y_n|\mathbf{w}, \mathbf{x}_n)}{\int p(\mathbf{w})\prod_n p(y_n|\mathbf{w}, \mathbf{x}_n)d\mathbf{w}}.$$

Unfortunately the integral in the denominator is intractable for this model. This is where variational inference comes to the rescue by approximating the posterior with a known form – in this case a Gaussian,

$$\begin{aligned} p(\mathbf{w}|\mathbf{X}, \mathbf{y}) &\approx q(\mathbf{w}), \\ &= \mathcal{N}(\mathbf{w}|\boldsymbol{\mu}, \boldsymbol{\Sigma}), \end{aligned}$$

where $\boldsymbol{\mu} \in \mathbb{R}^D$ and $\boldsymbol{\Sigma} \in \mathbb{R}^{D\times D}$. To make this approximation as close as possible, variational inference optimizes the Kullback Leibler divergence between this and true posterior using the evidence lower bound, ELBO, and the reparameterization trick in[1]:

$$\min_{\boldsymbol{\mu},\boldsymbol{\Sigma}} \text{KL}\left[q(\mathbf{w})\|p(\mathbf{w}|\mathbf{X}, \mathbf{y})\right].$$

One question you may ask is why would we want to go to all this bother over the MAP approach? Specifically, why learn an extra $\mathcal{O}(D^2)$ number of parameters over the MAP approach? Well, a few reasons, the first being that the weights are well regularised in this formulation, for instance we can actually learn ψ, rather than having to set it (this optimization of the prior is called empirical Bayes). Secondly, we have a principled way of incorporating modelling uncertainty over the weights into our predictions,

$$\begin{aligned} p(y^* = 1|\mathbf{X}, \mathbf{x}^*) &= \int \sigma(\mathbf{x}^{*\top}\mathbf{w})q(\mathbf{w})d\mathbf{w}, \\ &\approx \frac{1}{S}\sum_{s=1}^{S}\sigma(\mathbf{x}^{*\top}\mathbf{w}^{(s)}), \quad \mathbf{w}^{(s)} \sim q(\mathbf{w}). \end{aligned}$$

This will have the effect of making our predictive probabilities closer to 0.5 when the model is uncertain. The MAP approach has no mechanism to achieve this since it only learns the mode of the posterior, $\hat{\mathbf{w}}$, with no notion of variance.

So how do we implement this with Aboleth? Easy; we change `layers` to the following,

```
import numpy as np
import tensorflow as tf
import aboleth as ab

n_samples_ = tf.placeholder_with_default(5, [])
layers = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.DenseVariational(output_dim=1, std=1., full=True) >>
    ab.Activation(tf.nn.sigmoid)
)
```

[1] Kingma, D. P. and Welling, M. Auto-encoding variational Bayes. In ICLR, 2014.

Note we are using `DenseVariational` instead of `DenseMAP`. In the `DenseVariational` layer the `full` parameter tells the layer to use a full covariance Gaussian, and `std` is initial value of the weight prior standard deviation, ψ, which is optimized. Also we've set `n_samples=5` (as a default value of a place holder) in the `InputLayer`, this lets the subsequent layers know that we are making a *stochastic* model, that is, whenever we call `layers` we are actually expecting back 5 samples of the model output. This argument defaults to 1, which is why we got a one-dimensional 0th axis in the last section. In this instance a setting of 5 makes the `DenseVariational` layer multiply its input with 5 samples of the weights from the approximate posterior, $\mathbf{X}\mathbf{w}^{(s)}$, where $\mathbf{w}^{(s)} \sim q(\mathbf{w})$, for $s = \{1 \dots 5\}$. These 5 samples are then passed to the `Activation` layer. We have used a place holder here because we usually want to use more samples of the network for prediction than for training.

Then like before to complete the model specification:

```
net, kl = layers(X=X_)
likelihood = tf.distributions.Bernoulli(probs=net)
loss = ab.elbo(likelihood, Y_, N=10000, KL=kl)
```

The main differences here are that `reg` is now `kl`, and we use the `elbo` loss function. For all intents and purposes `kl` is still a regularizer on the weights (it is the Kullback Leibler divergence between the posterior and the prior distributions on the weights), and `elbo` is the evidence lower bound objective. Here `N` is the (expected) size of the dataset, we need to know this term in order to properly calculate the evidence lower bound when using mini-batches of data.

We train this model in exactly the same way as the logistic classifier, however prediction is slightly different - that is we need to average the samples drawn from the network to get a predicted probability (as in the sum over weight samples above),

```
predict_p = tf.reduce_mean(net, axis=0)
probs = net.eval(predict_p,
                 feed_dict={X_: X_query, n_samples_: 20})
```

So probs also has a shape of $(N^*, 1)$, and we have used 20 samples to calculate the average probability.

Note: If you used logits in the likelihood, then the prediction becomes:

```
predict_p = tf.reduce_mean(likelihood.probs, axis=0)
probs = net.eval(predict_p,
                 feed_dict={X_: X_query, n_samples_: 20})
```

7.2.3 Approximate Gaussian Processes

Aboleth also provides the building blocks to easily create scalable (approximate) Gaussian processes. We'll implement a simple Gaussian process regressor here, but for brevity, we'll skip the introduction to Gaussian processes, and refer the interested reader to[2].

The approximation we have implemented in Aboleth is the *random feature expansions* (see[3] and[4]), where we can approximate a kernel function from a set of random basis functions,

$$\mathrm{k}(\mathbf{x}_i, \mathbf{x}_j) \approx \frac{1}{S} \sum_{s=1}^{S} \phi^{(s)}(\mathbf{x}_i)^\top \phi^{(s)}(\mathbf{x}_j),$$

[2] Rasmussen, C. E., and Williams, C. K. I. Gaussian processes for machine learning. Cambridge: MIT press, 2006.
[3] Rahimi, A., & Recht, B. Random features for large-scale kernel machines. Advances in neural information processing systems. 2007.
[4] Cutajar, K. Bonilla, E. Michiardi, P. Filippone, M. Random Feature Expansions for Deep Gaussian Processes. In ICML, 2017.

with equality in the infinite limit. The trick is to find the right family of basis functions, ϕ, that corresponds to a particular family of kernel functions, e.g. radial basis, Matern, etc. This insight allows us to approximate a Gaussian process regressor with a *Bayesian linear regressor* using these random basis functions, $\phi^{(s)}(\mathbf{X})$!

We can easily do this using Aboleth, for example, with a radial basis kernel,

```
import tensorflow as tf
import aboleth as ab

lenscale = tf.Variable(1.)  # learn isotropic length scale
kern = ab.RBF(lenscale=ab.pos(lenscale))

n_samples_ = tf.placeholder_with_default(5, [])
layers = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.RandomFourier(n_features=100, kernel=kern) >>
    ab.DenseVariational(output_dim=1, full=True)
)
```

Here we have made `lenscale` a TensorFlow variable so it will be optimized, and we have also used the `ab.pos` function to make sure it stays positive. The `ab.RandomFourier` class implements random Fourier features[3], that can model shift invariant kernel functions like radial basis, Matern, etc. See *ab.kernels* for implemented kernels. We have also implemented random arc-cosine kernels[4] see `ab.RandomArcCosine` in *ab.layers*.

Then to complete the formulation of the Gaussian process (likelihood and loss),

```
std = tf.Variable(1.)  # learn likelihood std. deviation

net, kl = layers(X=X_)
likelihood = tf.distributions.Normal(net, scale=ab.pos(std))
loss = ab.elbo(likelihood, Y_, kl, N=10000)
```

Here we just have a Normal likelihood since we are creating a model for regression, and we can also get TensorFlow to optimise the likelihood standard deviation, `std`.

Training and prediction work in exactly the same way as the Bayesian logistic classifier. Here is an example of the approximate GP in action (see *Regression* for a more detailed demonstration);

7.2.4 See Also

For more detailed demonstrations of the functionality within Aboleth, we recommend you check out the demos,

- *Regression* and *SARCOS* - for more regression applications.
- *Multiple Input Data* - models with multiple input data types.
- *Bayesian Classification with Dropout* - Bayesian nets using dropout.
- *Imputation Layers* - let Aboleth deal with missing data for you.

7.2.5 References

7.3 Demos

We have included some demonstration scripts with Aboleth to help you get familiar with some of the possible model architectures that can be build with Aboleth. We also demonstrate in these scripts a few methods for actually training models using TensorFlow, and how to get up and running with TensorBoard, etc.

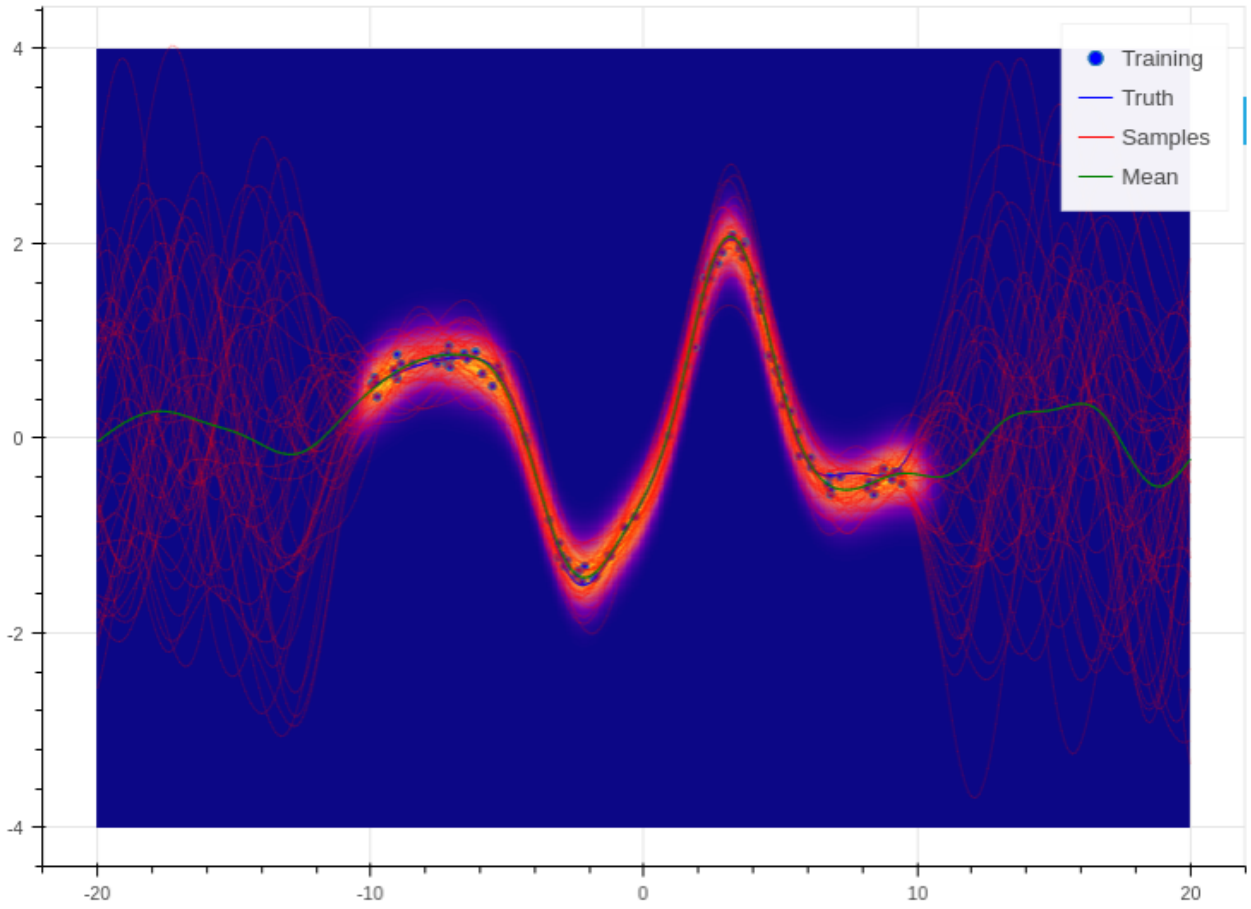
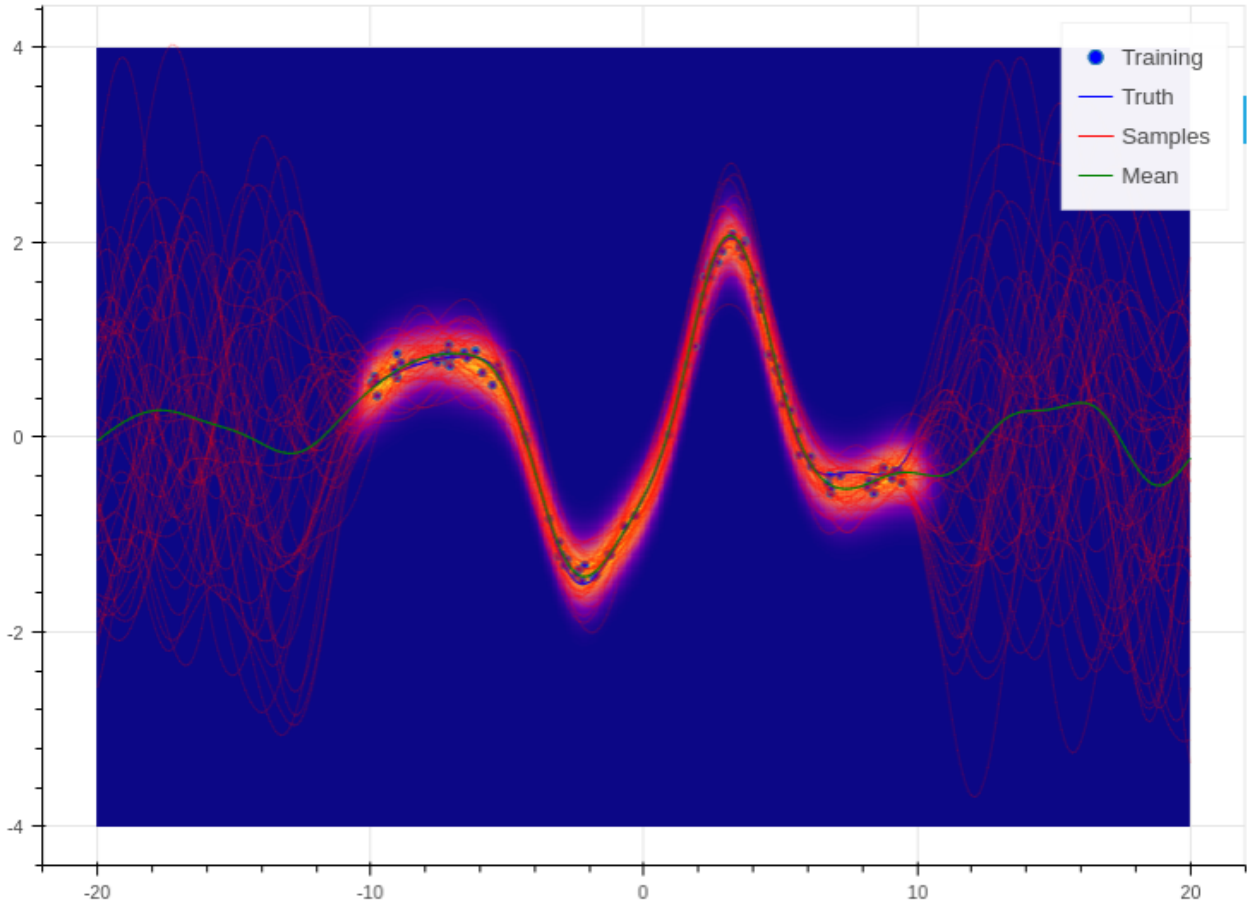


Fig. 7.1: Example of an approximate Gaussian process with a radial basis kernel. We have shown 50 samples of the predicted latent functions, the mean of these draws, and the heatmap is the probability of observing a target under the predictive distribution, $p(y^*|\mathbf{X}, \mathbf{y}, \mathbf{x}^*)$.

7.3.1 Regression

This is a simple demo that draws a random, non linear function from a Gaussian process with a specified kernel and length scale. We then use Aboleth (in Gaussian process approximation mode) to try to learn this function given only a few noisy observations of it. This script also demonstrates how we can divide the data into mini-batches using utilities in the tf.data module, and how we can use tf.train.MonitoredTrainingSession to log the learning progress.

This demo can be used to generate figures like the following:



You can find the full script here: regression.py.

7.3.2 SARCOS

Here we use Aboleth, again in Gaussian process regression mode, to fit the venerable SARCOS robot arm inverse kinematics dataset. The aim is to learn the inverse kinematics from 44484 observations of joint positions, velocities and accelerations to joint torques.

This problem is too large for a regular Gaussian process, and so is a good demonstration of why the approximation in Aboleth is useful (see *Approximate Gaussian Processes*). It also demonstrates how we learn automatic relevance determination (ARD, or anisotropic) kernels.

We have also demonstrated how you can use TensorBoard with the models you construct in Aboleth, so you can visually monitor the progress of learning. This also allows us to visualise the model's performance on the *validation* set every training epoch. Using TensorBoard has the nice side-effect of also enabling model check point saving, so you can actually *resume* learning this model if you run the script *again*!!

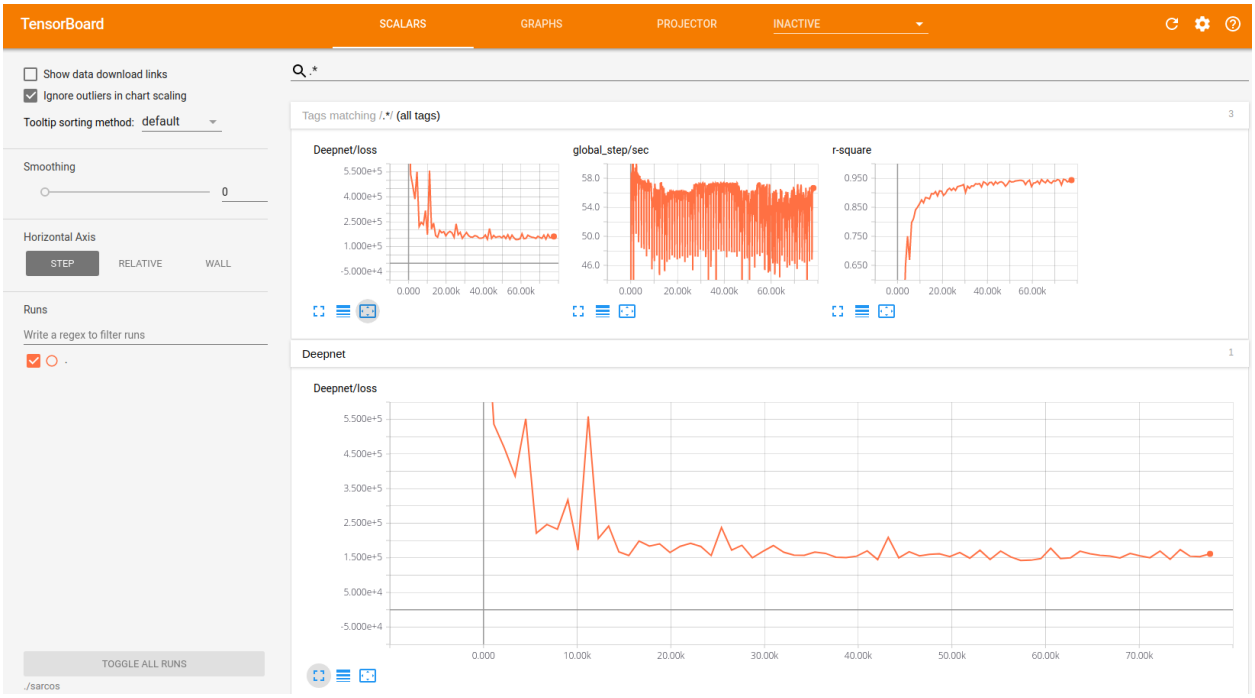


Fig. 7.2: Using TensorBoard to visualise the learning progress of the Aboleth model fitting the SARCOS dataset. The "r-square" plot here is made from evaluating the R-square performance on the held-out test set every epoch of training.

This demo will make a `sarcos` folder in the directory you run the demo from. This contains all of the model checkpoints, and to visualise these with TensorBoard, run the following:

```
$ tensorboard --logdir=./sarcos
```

The full script is here: sarcos.py.

7.3.3 Multiple Input Data

This demo takes inspiration from TensorFlow's Wide & Deep tutorial in that it treats continuous data separately from categorical data, though we combine both input types into a "deep" network. It also uses the census dataset from the TensorFlow tutorial.

We demonstrate a few things in this script:

- How to use Aboleth to learn embeddings of categorical data using the `ab.EmbedVariational` layer (see *ab.layers*).
- How to easily apply these embeddings over multiple columns using the `ab.PerFeature` higher-order layer (see *ab.hlayers*).
- Concatenating these input layers (using `ab.Concat`) before feeding them into subsequent layers to learn joint representations.
- How to loop over mini-batches directly using a `feed_dict` and an appropriate mini-batch generator, `ab.batch` (see *ab.util*).

Using this set up we get an accuracy of about 85.3%, compared to the wide and deep model that achieves 84.4%.

The full script is here: multi_input.py.

7.3.4 Bayesian Classification with Dropout

Here we demonstrate a slightly different take on Bayesian deep learning. Yarin Gal in his thesis and associate publications demonstrates that we can view regular neural networks with dropout as a form of variational inference with specific prior and posterior distributions on the weights.

In this demo we implement this elegant idea with maximum a-posteriori weight and dropout layers in a classifier (see *ab.layers*). We leave these layers as stochastic in the prediction step, and draw samples from the network's predictive distribution, as we would in variational networks.

We test the classifier against a random forest classifier on the breast cancer dataset with 5-fold cross validation, and get quite good and robust performance.

The script can be found here: classification.py

7.3.5 Imputation Layers

Aboleth has a few layers that we can use to *impute* data and also to *learn imputation statistics*, see *ab.impute*. This drastically simplifies the pipeline for dealing with messy data, and means our imputation methods can benefit from information contained in the labels (as opposed to imputing as a separate stage from supervised learning).

This script demonstrates an imputation layer that learns a "mean" and a "standard deviation" of a Normal distribution (per column) to *randomly* impute the data from! We compare it to just imputing the missing values with the column means.

The task is a multi-task classification problem in which we have to predict forest coverage types from 54 features or various types, described here. We have randomly removed elements from the features, which we impute using the two aforementioned techniques.

Naive mean imputation gives 68.7% accuracy (0.717 log loss), and the per-column Normal imputation gives 69.1% accuracy (0.713 log loss).

You can find the script here: imputation.py

7.3.6 Compatibility with TensorFlow / Keras

In most circumstances, Aboleth's layer composition framework is interoperable with TensorFlow and Keras layers. This gives us access to a vast range of layers not directly implemented in Aboleth which are suitable for various problems, such as LSTMs, GRUs and other variants of recurrent layers for sequence prediction, to name just one example.

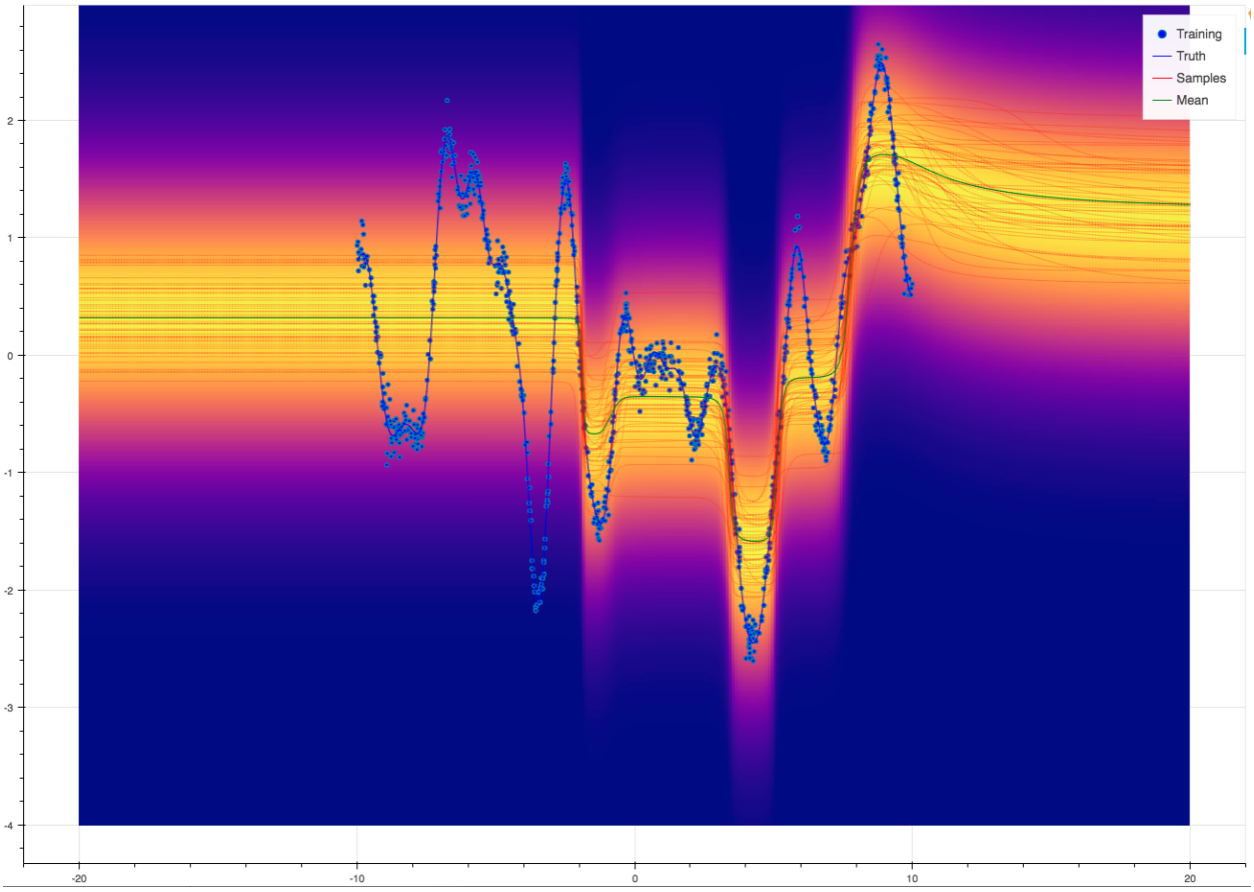
This script demonstrates how to use Keras dense layers with an Aboleth with dropout to approximate a Bayesian neural net. We also have a tutorial associated with the demo that you can find here: *Integrating Aboleth with Keras*.

You can find the script here: regression_keras.py

7.4 Authors

7.4.1 Development Leads

- Daniel Steinberg
- Lachlan McCalman
- Louis Tiao


Training
Truth
Samples
Mean
2
1
0
-1
-2
-3
-4
-20
-10
0
10
20

7.4.2 Contributors

- Simon O'Callaghan
- Alistair Reid
- Joyce Wang

7.5 Contributing Guidelines

Please contribute if you think a feature is missing in Aboleth, if you think an implementation could be better or if you can solve an existing issue!

We just request you read the following before making any changes to the codebase.

7.5.1 Pull Requests

This is the best way to contribute. We usually follow a git-flow based development cycle. The way this works is quite simple:

1. Make an issue on our github with your proposed feature or fix.
2. Fork, or make a branch name with the issue number like *feature/#113*.
3. When finished, submit a pull request to merge into *develop*, and refer to which issue is being closed in the pull request comment (i.e. *closes #113*).
4. One of use will review the pull request.
5. If accepted, your feature will be merged into develop.
6. Your change will eventually be merged into master and tagged with a release number.

7.5.2 Code and Documentation Style

In Aboleth we are *only* targeting python 3 - our code is much more elegant as a result, and we don't have the resources to also support python 2, sorry.

We adhere to the PEP 8 convention for code, and the PEP 257 convention for docstrings, using the NumPy/SciPy documentation style. Our continuous integration automatically runs linting checks, so any pull-request will automatically fail if these conventions are not followed.

The builtin Sphinx extension Napoleon is used to parse NumPy style docstrings. To build the documentation you can run `make` from the `docs` directory with the html option:

```
$ make html
```

7.5.3 Testing

We use py.test for all of our unit testing, most of which lives in the `tests` directory, with *judicious* use of doctests – i.e. only when they are illustrative of a functions usage.

All of the dependencies for testing can be installed by issuing:

```
$ pip install -e .[dev]
```

You can run the tests by issuing from the top level repository directory:

```
$ pytest .
```

Our continuous integration (CI) will fail if coverage drops below 90%, and we generally want coverage to remain significantly above this. Furthermore, our CI will fail if the code doesn't pass PEP 8 and PEP 257 conventions. You can run the exact CI tests by issuing:

```
$ make coverage
$ make lint
```

from the top level repository directory.

7.6 Tutorials

The following are guided tutorials on how to use Aboleth for particular machine learning tasks. These are a great place to start getting more familiar with how to use Aboleth for more complex problems.

7.6.1 Saving and Loading Aboleth Models

In this tutorial we will cover the basics of how to save and load models constructed with Aboleth. We don't provide any inherent saving and loading code in this library, and rely directly on TensorFlow functionality.

Naming the Graph

Even though the whole graph you create is saved and automatically named, it helps when loading to know the exact name of the part of the graph you want to evaluate. So to begin, we will create a very simple Bayesian linear regressor with place holders for data. Let's start with the place holders,

```
with tf.name_scope("Placeholders"):
    n_samples_ = tf.placeholder_with_default(NSAMPLES, shape=[],
                                             name="samples")
    X_ = tf.placeholder_with_default(X_train, shape=(None, D),
                                     name="X")
    Y_ = tf.placeholder_with_default(Y_train, shape=(None, 1),
                                     name="Y")
```

We have used a `name_scope` here for easy reference later. Also, we'll assume variables in all-caps have been defined elsewhere. Now let's make our simple network (just a linear layer),

```
net = ab.stack(
    ab.InputLayer(name='X', n_samples=n_samples_),
    ab.DenseVariational(output_dim=1, full=True)
)
```

And now lets build and name our graph and associate names with the parts of it we will to evaluate later,

```
with tf.name_scope("Model"):
    f, kl = net(X=X_)
    likelihood = tf.distributions.Normal(loc=f, scale=ab.pos(NOISE))
    loss = ab.elbo(likelihood, Y_, N, kl)

with tf.name_scope("Predict"):
```

```
    tf.identity(f, name="f")
    ab.sample_mean(f, name="Ey")
```

Now note how we have used `tf.identity` here to name the latent function, `f`, again this is so we can easily load it later for drawing samples from our network. We also don't need any variables to assign these operations to (unless we want to use them before saving), we just need to build them into the graph.

Saving the Graph

At this point we recommend reading the Tensorflow tutorial on saving and restoring. We typically use a `tf.MonitoredTrainingSession` as it handles all of the model saving and check-pointing etc. You can see how we do this in the *SARCOS* demo, but we have also copied the code below for convenience,

```
# Training graph with step counter
with tf.name_scope("Train"):
    optimizer = tf.train.AdamOptimizer()
    global_step = tf.train.create_global_step()
    train = optimizer.minimize(loss, global_step=global_step)

# Logging
log = tf.train.LoggingTensorHook(
    {'step': global_step, 'loss': loss},
    every_n_iter=1000
)

# Training loop
with tf.train.MonitoredTrainingSession(
        config=CONFIG,
        checkpoint_dir="./",
        save_summaries_steps=None,
        save_checkpoint_secs=20,
        save_summaries_secs=20,
        hooks=[log]
) as sess:
    for i in range(NEPOCHS):

        # your training code here
        ...
```

This code will also make it easy to use TensorBoard to monitor your training, simply point it at the `checkpoint_dir` and run it like,

```
$ tensorboard --logdir=<checkpoint_dir>
```

Once you are satisfied that your model has converged, you can just kill the python process. If you think it could do with a bit more "baking", then just simply re-run the training script and the `MonitoredTrainingSession` will ensure you resume learning where you left off!

Loading Specific Parts of the Graph for Prediction

Typically we only want to evaluate particular parts of the graph (that is, the ones we named previously). In this section we'll go through how to load the last checkpoint saved by the `MonitoredTrainingSession`, and to get hold of the tensors that we named. We then use these tensors to predict on new query data!

```
# Get latest checkpoint
model = tf.train.latest_checkpoint(CHECKPOINT_DIR)

# Make a graph and a session we will populate with our saved graph
graph = tf.Graph()
with graph.as_default():

    sess = tf.Session()
    with sess.as_default():

        # Restore graph
        saver = tf.train.import_meta_graph("{}.meta".format(model))
        saver.restore(sess, model_file)

        # Restore place holders
        X_ = graph.get_operation_by_name("Placeholders/X").outputs[0]
        Y_ = graph.get_operation_by_name("Placeholders/Y").outputs[0]
        n_samples_ = graph.\
            get_operation_by_name("Placeholders/samples").outputs[0]

        feed_dict = {X_: X_test, n_samples_: PREDICTSAMPLES}

        f = graph.get_operation_by_name("Predict/f").outputs[0]
        Ey = graph.get_operation_by_name("Predict/Ey").outputs[0]

        f_samples, y_pred = sess.run([f, Ey], feed_dict=feed_dict)
```

The most complicated part of the above code is remembering all of the boiler-plate to insert the saved graph into a new session, and then do get our place holders and prediction tensors. Once we have done this though, evaluating the operations we need for prediction is handled in the usual way. We have also assumed in this demo that you want to use more samples for prediction (`PREDICTSAMPLES`) than for training (`NSAMPLES`), so we have made this also a place holder.

That's it!

7.6.2 A Regression Master Class with Aboleth

In this tutorial we will show you how to build a variety of linear and non linear regressors with the building blocks in Aboleth - and demonstrate how easy it is once you have the basics down!

We'll start off with with some linear regressors, then we'll extend these models to various types of neural networks. We'll also talk about how we can approximate other types of non linear regressors with Aboleth, such as support vector regressors and Gaussian processes.

Firstly, for the purposes of this tutorial we have generated 100 noisy samples from the non-linear function,

$$y_i = \frac{\sin(x_i)}{x_i} + \epsilon_i,$$

where we draw $\epsilon_i \sim \mathcal{N}(0, 0.05)$. We will use this data to fit the regressors, with the aim of getting them to reconstruct the latent function,

$$f = \frac{\sin(x)}{x},$$

with as little error as possible. This is what this data set looks like:

We use R^2, AKA the coefficient of determination to evaluate how good the estimate of the latent functions is. An R^2 of 1.0 is a perfect fit, and 0.0 means no better than a Normal distribution fit only to the targets, y_i.

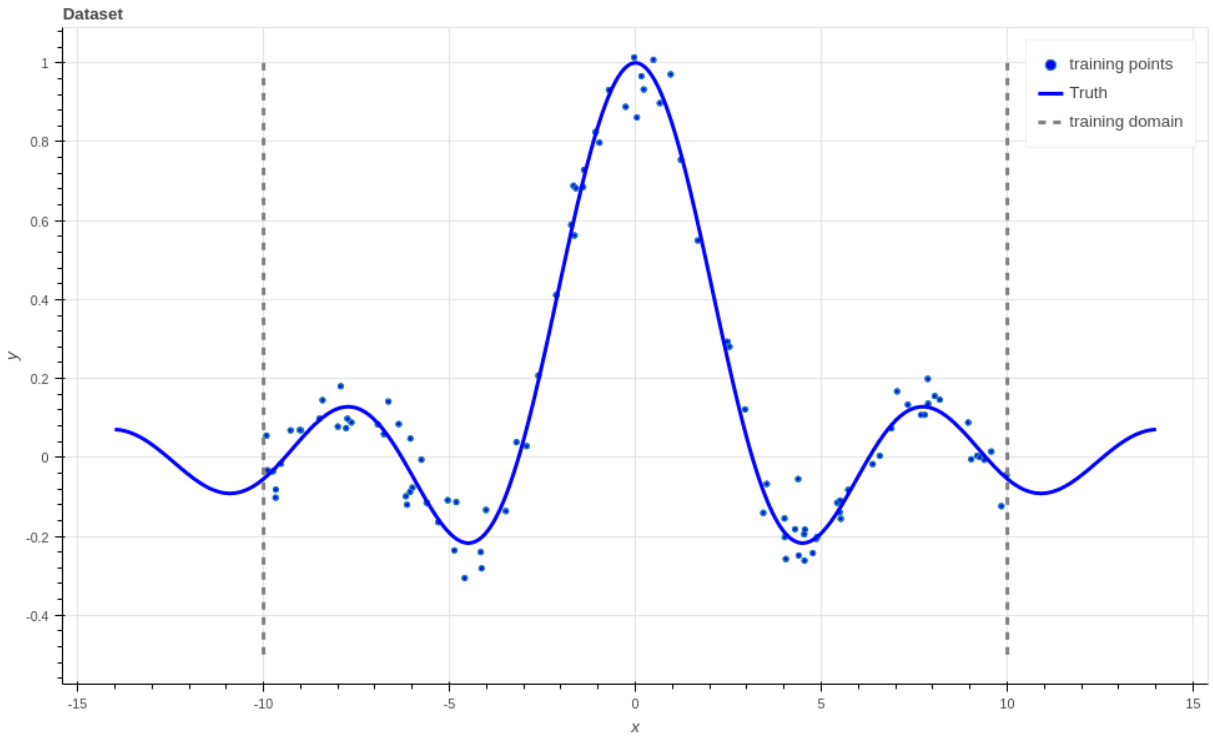


Fig. 7.3: The dataset used for fitting the regressors. There are 100 noisy training points (blue dots) that the algorithms get to see, and 1000 noise free points (blue line) that the algorithm has to predict.

Note in the figure above that we have only generated training data for x from -10 to 10, but we evaluate the algorithms from -14 to 14. This is because we want to see how well the algorithms extrapolate away from the data; which is a hard problem. We don't evaluate the R^2 in this extrapolation region since it makes it harder to differentiate the performance of the algorithms within the bounds of the training data. However, it is interesting to see how the algorithms perform in this region.

Linear regression

The easiest algorithms to build with Aboleth are linear regressors, and so this is where we'll start this tutorial. Specifically, we'll start with ridge regression, which represents the latent function as,

$$f = wx + b,$$

where w and b are the regression weights and bias we wish to learn. It has the following objective function ($l2$ "reconstruction error" loss and $l2$ regularisation),

$$\min_{w,b} \frac{1}{2N} \sum_{i=1}^{N} \|wx_i + b - y_i\|_2^2 + \frac{\lambda}{2} \left(\|w\|_2^2 + \|b\|_2^2 \right),$$

where λ is the regularization coefficient that penalises large magnitude weights. This can be simply implemented in Aboleth using the following code,

```
lambda_ = 1e-4  # Weight regularizer
noise = 1.  # Likelihood st. dev.

net = (
```

```
    ab.InputLayer(name="X") >>
    ab.DenseMAP(output_dim=1, l2_reg=lambda_, l1_reg=0.)
)

f, reg = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=noise)
loss = ab.max_posterior(lkhood, Y, reg)
```

Here `reg` is the second regularizing term in the objective function, and putting a `Normal` likelihood distribution with a standard deviation of 1.0, gives us the first term, up to a constant value, when using `max_posterior` (we are performing *maximum a-posteriori* inference here, hence the name of the function). Alternatively, if we didn't want to use a likelihood function we could have constructed the `loss` as

```
loss = 0.5 * tf.reduce_mean((Y - f)**2) + reg
```

We can then give this `loss` tensor to a TensorFlow optimisation routine, such as the `AdamOptimizer` that we use in the regression demo.

When we evaluate the model (the tensor `f`) on the testing inputs we get a terrible result:

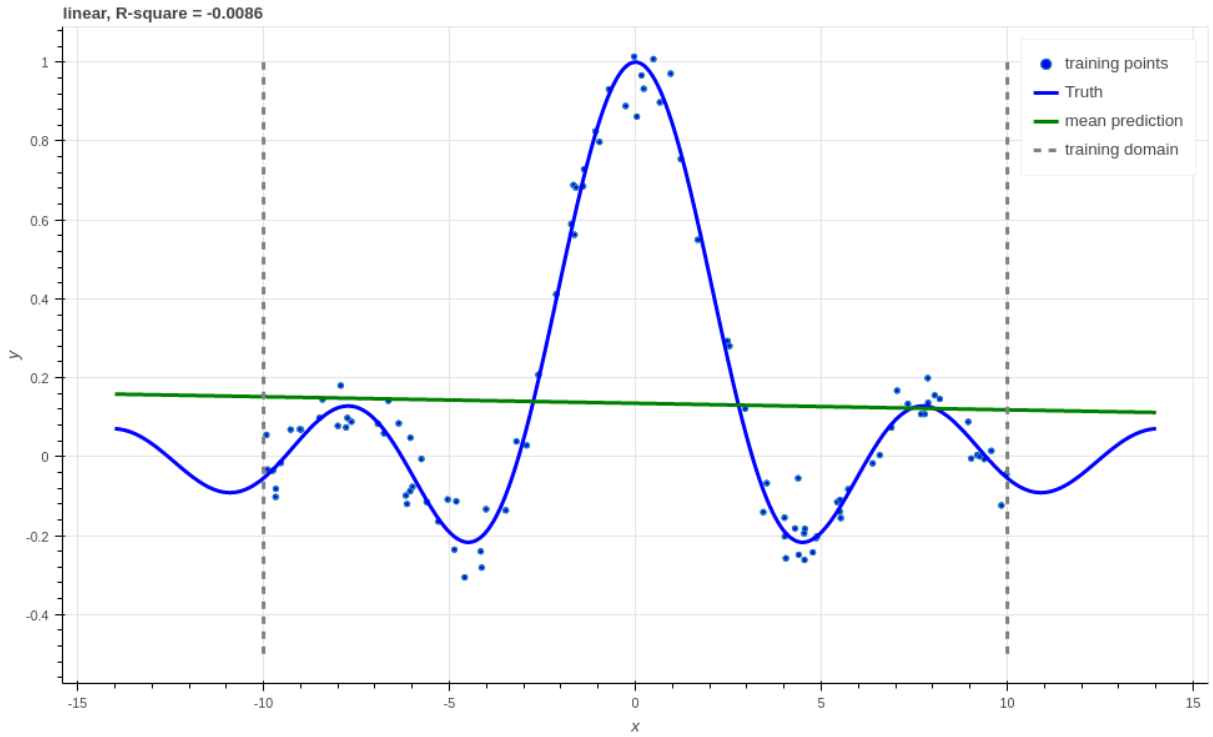


Fig. 7.4: Ridge linear regression, R-square ≈ 0.

Which we would expect from fitting a linear regressor to a non-linear function! Just for illustrative purposes we'll now make a Bayesian linear regressor. We shouldn't expect this to do any better than the ridge regressor since they have equivalent predictive means. However, it is not really any harder to create this regressor using Aboleth, and we can also easily obtain predictive uncertainty from it.

In a Bayesian linear regressor (following[1]) we model the observations as being drawn from a Normal likelihood, and

[1] Rasmussen, C.E., and Williams, C.K.I. "Gaussian processes for machine learning." Vol. 1. Cambridge: MIT press, 2006.

the weights from a Normal prior (we have ignored the bias, b, for simplicity),

$$y_i \sim \mathcal{N}(wx_i, \sigma^2),$$
$$w \sim \mathcal{N}(0, \lambda^{-1}),$$

where the aim is to estimate the parameters of the *posterior* distribution over the weights (and optionally point estimates for λ, σ),

$$w \sim \mathcal{N}(m, v).$$

The objective we use in this case is the *evidence lower bound* or *ELBO* as it is easy to use with stochastic gradient descent for a variety of different models[3]. For Bayesian linear regression the ELBO takes the form,

$$\min_{m,v,\sigma,\lambda} -\sum_{i=1}^{N} \mathbb{E}_{\mathcal{N}(w|m,v)}\left[\log \mathcal{N}(y_i|wx_i, \sigma^2)\right] + \mathrm{KL}\left[\mathcal{N}(w|m,v) \| \mathcal{N}(w|0, \lambda^{-1})\right].$$

This looks complicated, but it's actually not too bad, especially when we compare it to the ridge regression objective. Firstly, the expectation acts like a data-fitting term (expected log likelihood of the targets given the inputs and the weights), which corresponds to the l2 reconstruction term. Next, the Kullback Leibler term (KL) is acting as a regulariser on the weights, penalizing the posterior diverging from the prior. We can implement this model with Aboleth using the following code,

```
lambda_ = 100.
std = (1 / lambda_) ** .5  # Weight st. dev. prior
noise = tf.Variable(1.)  # Likelihood st. dev. initialisation

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.DenseVariational(output_dim=1, std=std, full=True)
)

f, kl = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=ab.pos(noise))
loss = ab.elbo(lkhood, Y, N, kl)
```

Note here that we have set `n_samples_` to some value (e.g. 5, or use a place holder) because the `DenseVariational` layer uses samples from its posterior distribution on the weights for evaluation. The more samples, the smoother the estimates of the model gradients during training, and the better the estimate of the posterior predictive distribution when querying (more on this soon).

Again, since we're using a linear model, we don't get great performance.

What's the point of going to all this effort implementing the ELBO over just the ridge regression? Well a few reasons, firstly we can use this objective to estimate the parameters σ & λ (this is called empirical Bayes, see[2] Section 3.5 for a good explanation). Secondly, since we have a posterior distribution over w, we can get a distribution over predictions of the latent function, f – samples from which we can see in the above figure. This tells us how confident out model is in its predictions. This will come in handy later with some of the more complex models.

Note: The model looks over-confident in its estimation of observations, however, we have only sampled the *latent function*. The value learned for the likelihood standard deviation, σ, is quite large, and compensates for this small latent function variance.

If we wanted to obtain predictive samples from our model over the *observations* instead of just the *latent function*, we would simply need to draw samples from our likelihood (e.g. `lkhood.sample()`).

Ok, now lets move beyond building linear models with Aboleth.

[3] Kingma, D. P. and Welling, M. "Auto-encoding variational Bayes." In ICLR, 2014.
[2] Bishop, C. M. "Pattern recognition and machine learning." Springer, 2006.

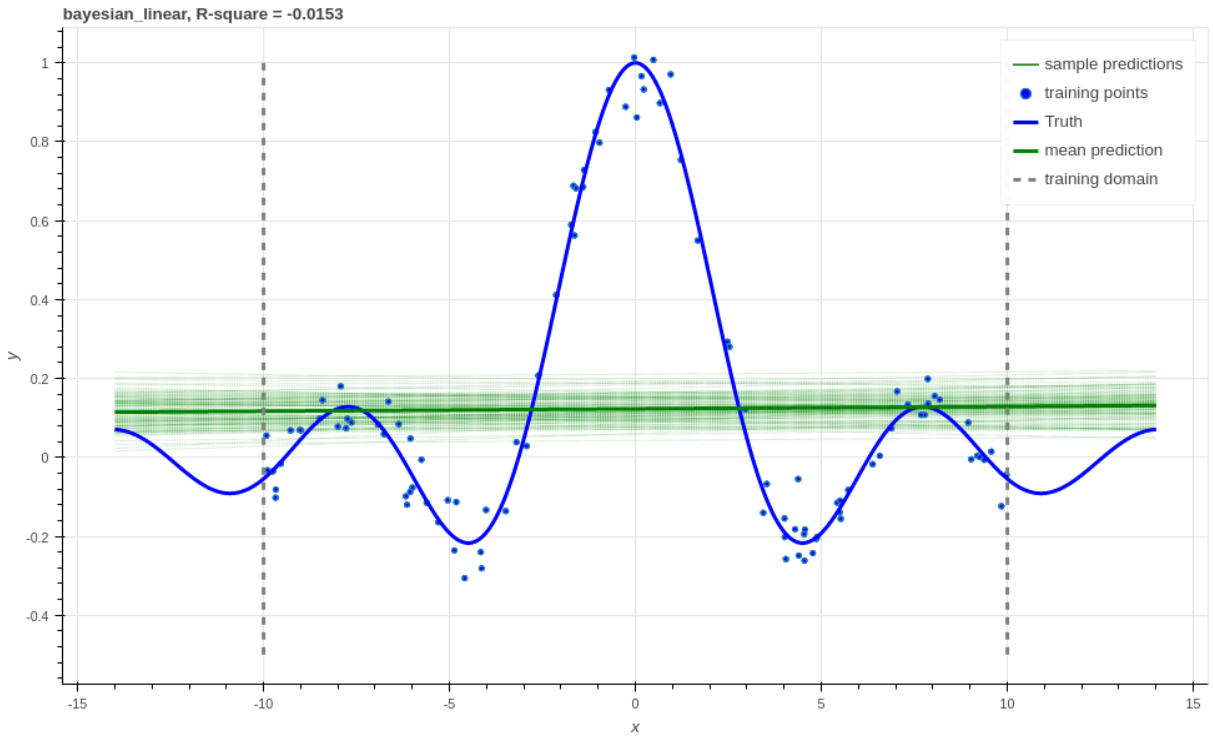


Fig. 7.5: Bayesian linear regression, R-square ≈ 0.

Neural Networks

The first family of non-linear regressors we'll look at now are neural networks and represent the latent function as,

$$f = \text{NN}(x).$$

Here NN refers to the neural net function, which is a sequential composition of linear layers (like our linear regressor) and non-linear activation functions. Learning a neural net classically has an objective something like,

$$\min_{w,b} \frac{1}{2N} \sum_{i=1}^{N} \|\text{NN}(x_i) - y_i\|_2^2 + \sum_{l=1}^{L} \frac{\lambda_l}{2} \left(\|w_l\|_2^2 + \|b_l\|_2^2 \right).$$

Note that it also has regularisers for each of the L linear layers in the network.

In this tutorial we use 4 layers, and the code for constructing this model in Aboleth is here:

```
lambda_ = 1e-4  # Weight regularizer
noise = .5  # Likelihood st. dev.

net = (
    ab.InputLayer(name="X", n_samples=1) >>
    ab.DenseMAP(output_dim=40, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DenseMAP(output_dim=20, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DenseMAP(output_dim=10, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DenseMAP(output_dim=1, l2_reg=lambda_, l1_reg=0.)
)
```

```
f, reg = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=noise)
loss = ab.max_posterior(lkhood, Y, reg)
```

Where we have used hyperbolic tan activation functions. Now we get much better performance on our regression task!

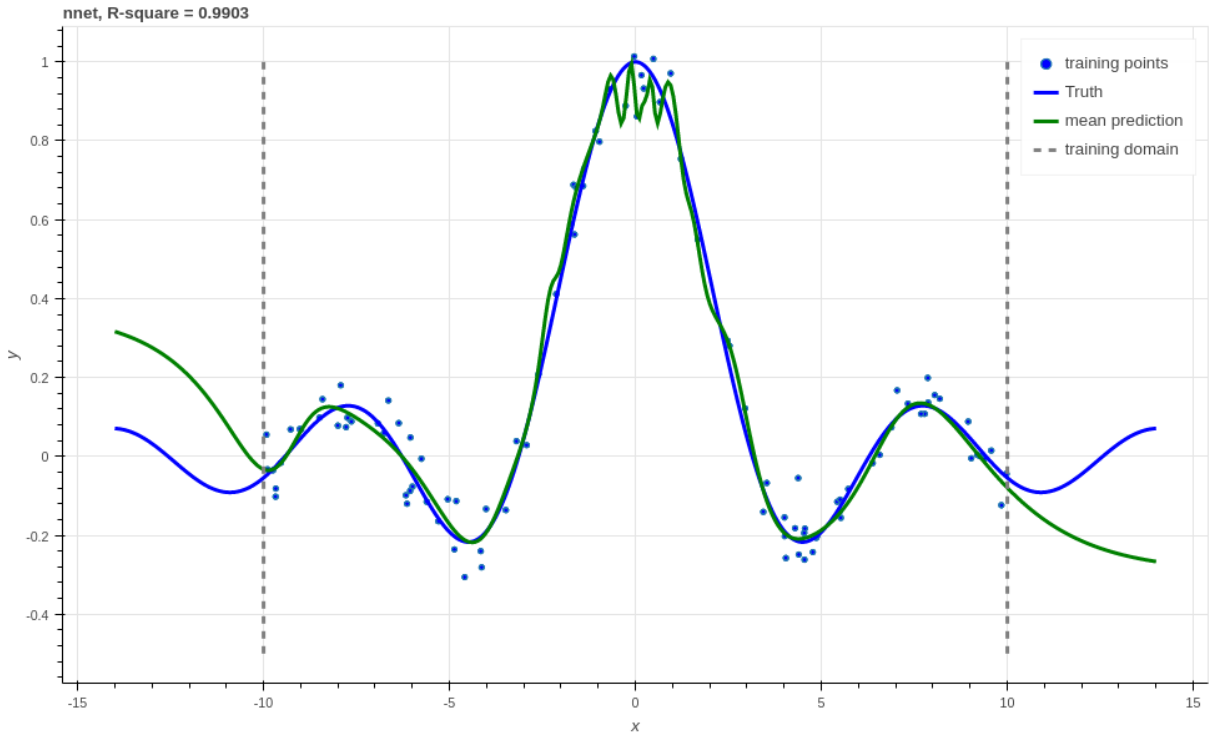


Fig. 7.6: Neural network with l2 regularization, R-square 0.9903.

There is a very easy trick to turn the above network into a Bayesian neural net, courtesy of Yarin Gal[4]. All we have to do is to add dropout to our network, and then keep dropout on during prediction! We can optionally also sample the network more than once during learning since the dropout makes it a stochastic network like our variational layers.

```
lambda_ = 1e-3  # Weight prior
noise = .5  # Likelihood st. dev.

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.DenseMAP(output_dim=40, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DropOut(keep_prob=0.9) >>
    ab.DenseMAP(output_dim=20, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DropOut(keep_prob=0.95) >>
    ab.DenseMAP(output_dim=10, l2_reg=lambda_, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DenseMAP(output_dim=1, l2_reg=lambda_, l1_reg=0.)
)
```

[4] Gal, Yarin. "Uncertainty in deep learning." PhD thesis, University of Cambridge, 2016.

```
f, reg = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=noise)
loss = ab.max_posterior(lkhood, Y, reg)
```

Now we get uncertainty on our latent functions:

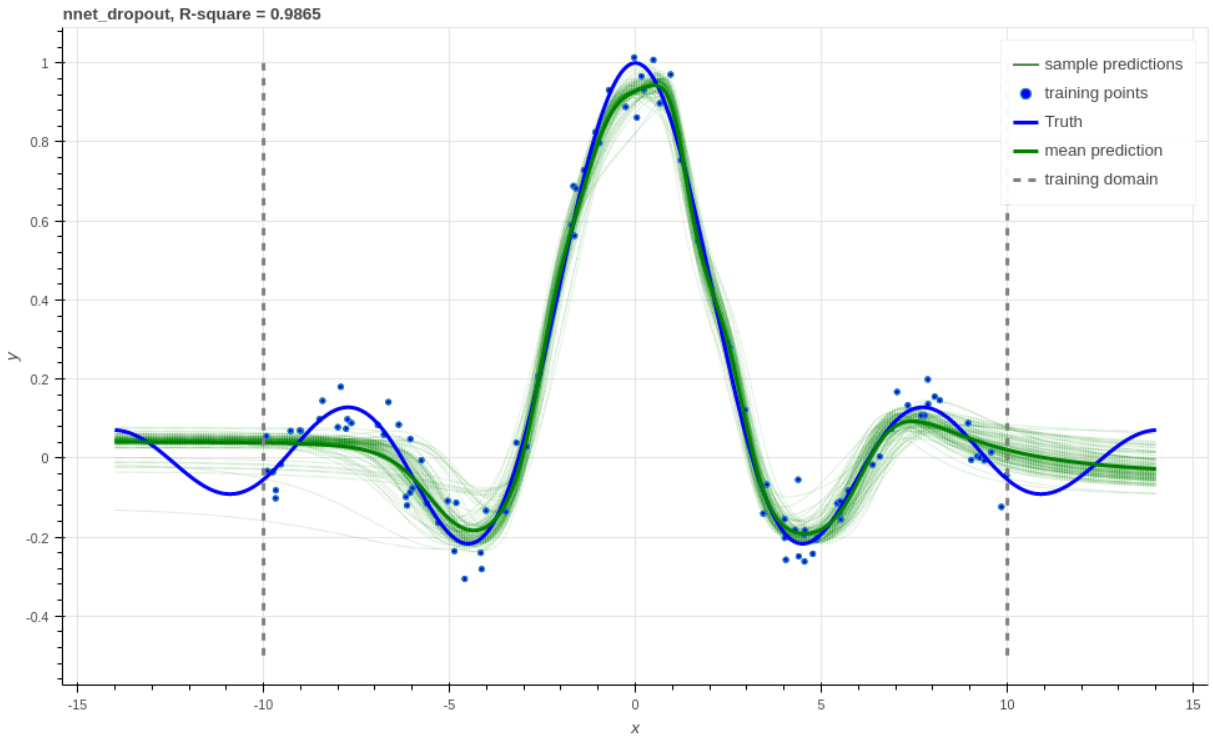


Fig. 7.7: Neural network with dropout, R-square 0.9865.

Though in this example we have a smoother prediction than the regular neural network and have lost a bit of performance… this is something we could potentially rectify with a bit more architecture tweaking (tuning the regularisers per layer for example).

We can also use our `DenseVariational` layers with an *ELBO* objective to create a Bayesian neural net. For brevity's sake we won't go into the exact form of the objective, except to say that it parallels the conversion of the linear regressor objective to the neural network objective. The code for building the Bayesian neural net regressor is,

```
lambda_ = 1e-1  # Weight prior
noise = tf.Variable(0.01)  # Likelihood st. dev. initialisation

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.DenseVariational(output_dim=20, std=lambda_) >>
    ab.Activation(tf.nn.relu) >>
    ab.DenseVariational(output_dim=7, std=lambda_) >>
    ab.Activation(tf.nn.relu) >>
    ab.DenseVariational(output_dim=5, std=lambda_) >>
    ab.Activation(tf.tanh) >>
    ab.DenseVariational(output_dim=1, std=lambda_)
)

f, kl = net(X=X)
```

```
lkhood = tf.distributions.Normal(loc=f, scale=ab.pos(noise))
loss = ab.elbo(lkhood, Y, N, kl)
```

Unfortunately, this prediction is even smoother than the previous one. This behaviour with Gaussian weight distributions is also something observed in[4], and is likely because of the strong complexity penalty coming from the KL regulariser.

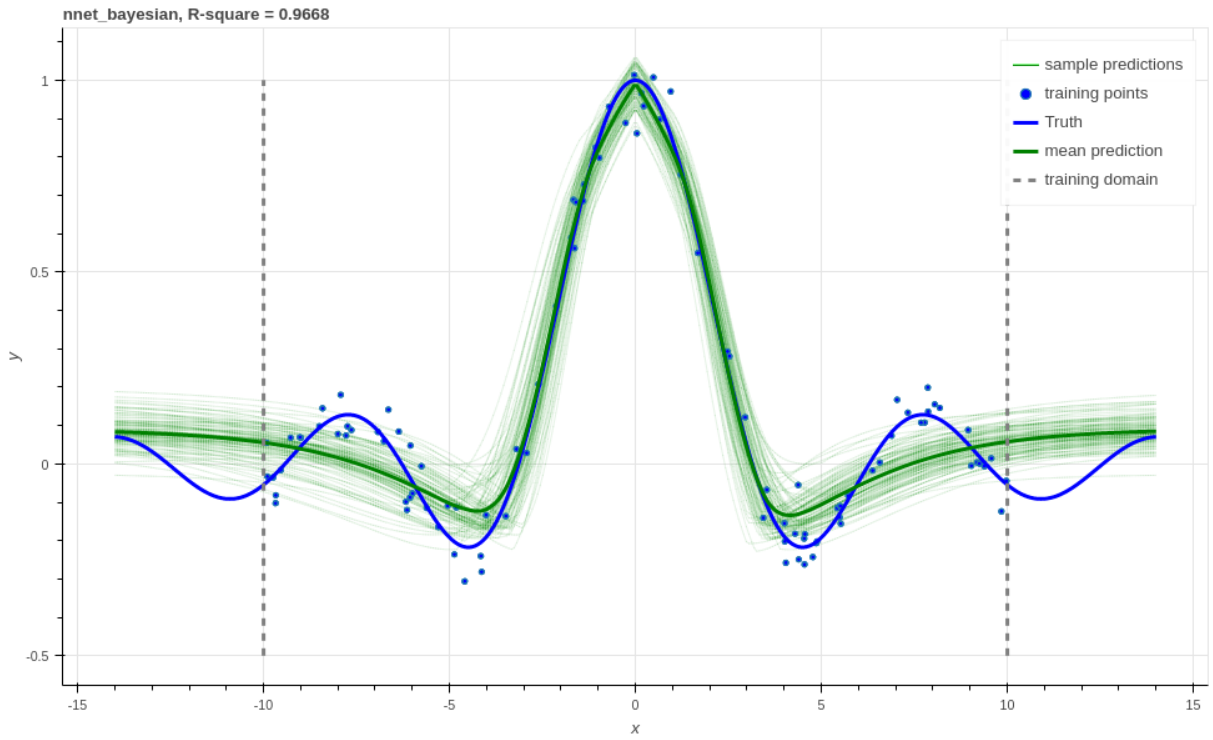


Fig. 7.8: Bayesian Neural network, R-square 0.9668.

If we train with more data, like in the figure below that uses 1000 training points as opposed to 100, the KL term has less of an influence and we obtain a good fit – at least inside the range of the training data. This suggests that with these types of Bayesian neural networks we need a lot of data to justify fitting a complex function (or fewer parameters).

Support Vector-like Regression

We can also approximate a non linear support vector regressor (SVR) with Aboleth. This approximation represents the latent function as,

$$f = w \times \text{RFF}(x) + b$$

Where RFF are random Fourier features[5], that approximate the radial basis functions used in kernel support vector machines. We learn the parameters using the following objective,

$$\min_{w,b} \frac{1}{N} \sum_{i=1}^{N} \max\left(|w \times \text{RFF}(x_i) + b - y_i| - \epsilon, 0\right) + \frac{\lambda}{2}\left(\|w\|_2^2 + \|b\|_2^2\right),$$

were $\epsilon \geq 0$ is the SVR's threshold parameter, under which errors go un-penalised. Naturally we will be using stochastic gradients to solve this objective, and not the original convex SVR formulation. Despite these approximations, we would expect support vector regressor-like behaviour! The code for this is as follows:

[5] Rahimi, Ali, and Benjamin Recht. "Random features for large-scale kernel machines." In NIPS, 2007.

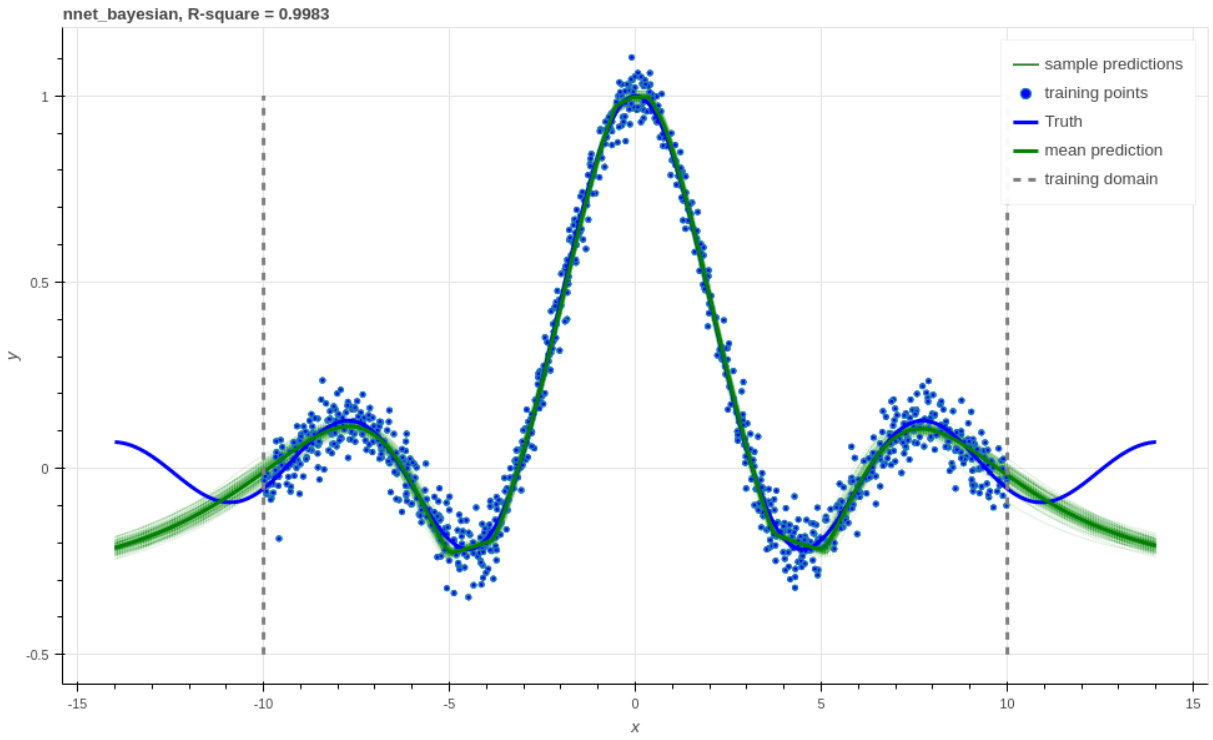


Fig. 7.9: Bayesian Neural network with 1000 training points, R-square 0.9983.

```
lambda_ = 1e-4
eps = 0.01
lenscale = 1.

# Specify kernel to approximate with the random Fourier features
kern = ab.RBF(lenscale=lenscale)

net = (
    ab.InputLayer(name="X", n_samples=1) >>
    ab.RandomFourier(n_features=50, kernel=kern) >>
    ab.DenseMAP(output_dim=1, l2_reg=lambda_, l1_reg=0.)
)

f, reg = net(X=X)
loss = tf.reduce_mean(tf.nn.relu(tf.abs(Y - f) - eps)) + reg
```

This results in the following prediction, which is the best we have achieved so far (not including the 1000 training point Bayesian neural net). Though its extrapolation performance leaves quite a lot to be desired.

Interestingly, because Aboleth is just a set of "building blocks" we can employ the same dropout trick that we used previously to make a "Bayesian" support vector regressor. We just insert a `DropOut` layer after the `RandomFourier` layer in the code above and increase the number of samples, this gives the following prediction.

This is better than our last SVR prediction, and adding the dropout layer seems to have somewhat controlled our extrapolation problem.

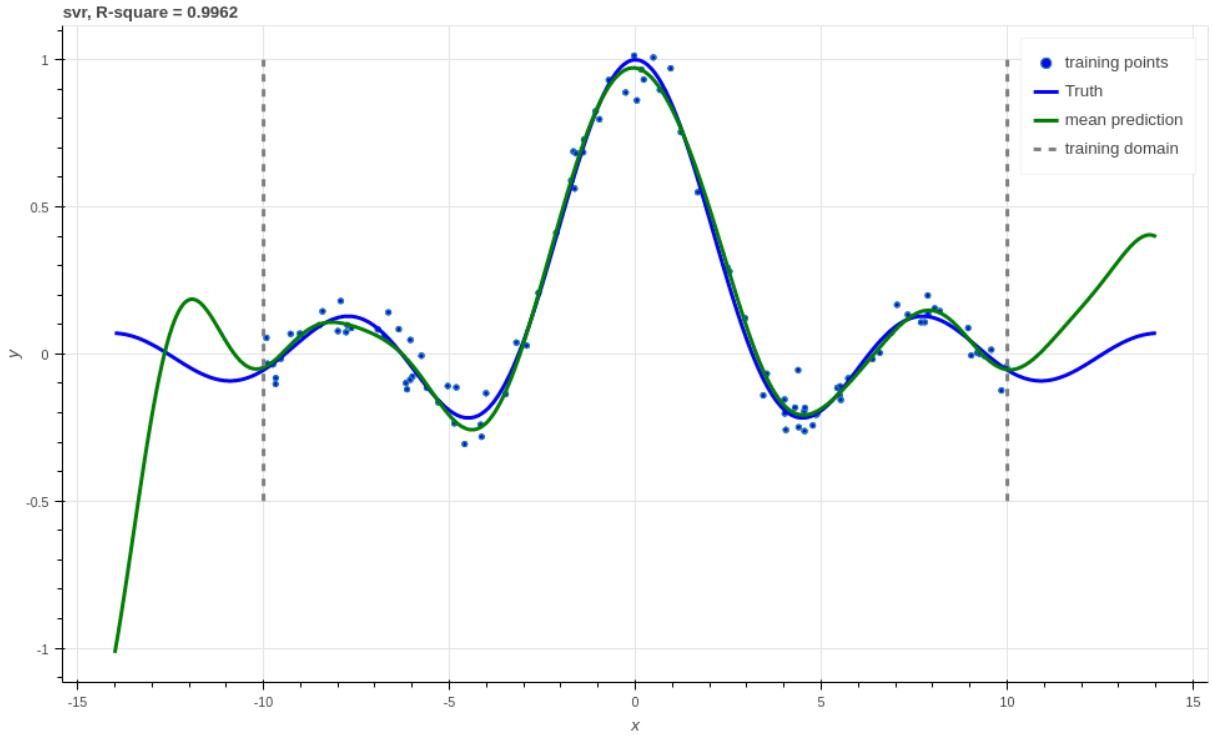


Fig. 7.10: Support vector regression, R-square 0.9962.

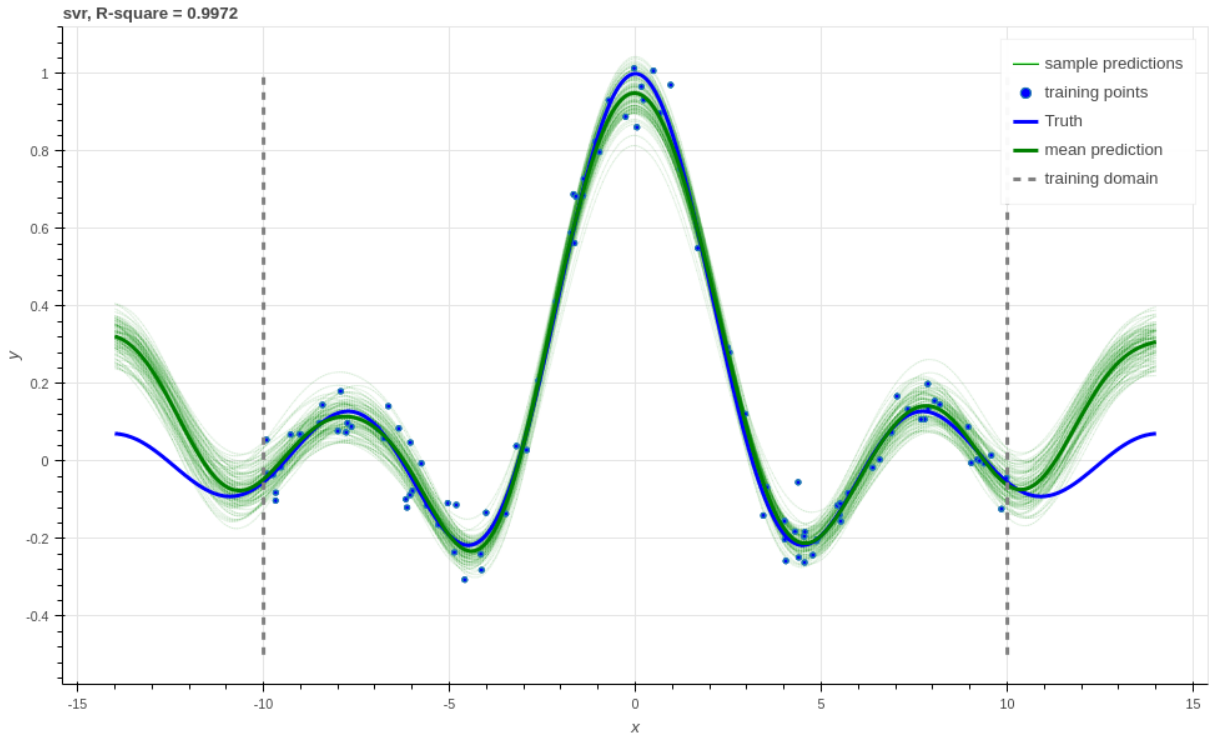


Fig. 7.11: Support vector regression with dropout, R-square 0.9972.

Gaussian process

The final class of non-linear regressors we will construct with Aboleth are (approximate) Gaussian process (GP) regressors. They represent the latent function in a similar manner to SVRs, but have a different learning objective. See[1] for a full discussion and derivation of GPs, we'll not go into detail in this tutorial.

Full Gaussian processes have a computational complexity of $\mathcal{O}(N^3)$ in training where N is the training set size. This limits their application to fairly small problems; a few thousands of training points. However, again using random Fourier features[5], we can approximate them by slightly modifying the Bayesian linear regressor from before,

```
lambda_ = 0.1  # Initial weight prior std. dev
noise = tf.Variable(.5)  # Likelihood st. dev. initialisation
lenscale = tf.Variable(1.)  # learn the length scale
kern = ab.RBF(lenscale=ab.pos(lenscale))  # keep length scale +ve

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.RandomFourier(n_features=50, kernel=kern) >>
    ab.DenseVariational(output_dim=1, std=lambda_, full=True)
)

f, kl = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=ab.pos(noise))
loss = ab.elbo(lkhood, Y, N, kl)
```

Which makes these approximate GPs scale linearly with N and allows us to trivially use mini-batch stochastic gradient optimisation! The tradeoff is, of course, how well they approximate GPs (in much the same way using random Fourier features approximated SVRs before).

When we look at our prediction, we can see that we can approximate a GP pretty well, and we get the sensible extrapolation behaviour we would expect from a GP too - falling back to zero away from the data in this case. Though, perhaps it over-estimates the uncertainty in the latent function relative to a regular GP. And as expected, the GP performs similarly to the "Bayesian" SVR in terms of R^2 within the training domain.

Finally, we can also easily implement some of the recent Fourier feature Deep-GP algorithms with Aboleth, such as those presented in[6]:

```
lambda_ = 0.1  # Initial weight prior std. dev
noise = tf.Variable(.01)  # Likelihood st. dev. initialisation
lenscale = tf.Variable(1.)  # learn the first length scale only

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.RandomFourier(n_features=20, kernel=ab.RBF(ab.pos(lenscale))) >>
    ab.DenseVariational(output_dim=5, std=lambda_, full=False) >>
    ab.RandomFourier(n_features=10, kernel=ab.RBF(1.)) >>
    ab.DenseVariational(output_dim=1, std=lambda_, full=False)
)

f, kl = net(X=X)
lkhood = tf.distributions.Normal(loc=f, scale=ab.pos(noise))
loss = ab.elbo(lkhood, Y, N, kl)
```

On such a simple problem we obtain similar performance to the regular GP, though we see that extrapolation is worse, and is quite reminiscent of the Neural network and SVR behaviour we were seeing previously. It would be interesting to explore why this happens, and if it is a consequence of the variational approximation, the random Fourier features, or just an inherent property of Deep-GPs.

[6] Cutajar, K. Bonilla, E. Michiardi, P. Filippone, M. "Random Feature Expansions for Deep Gaussian Processes." In ICML, 2017.

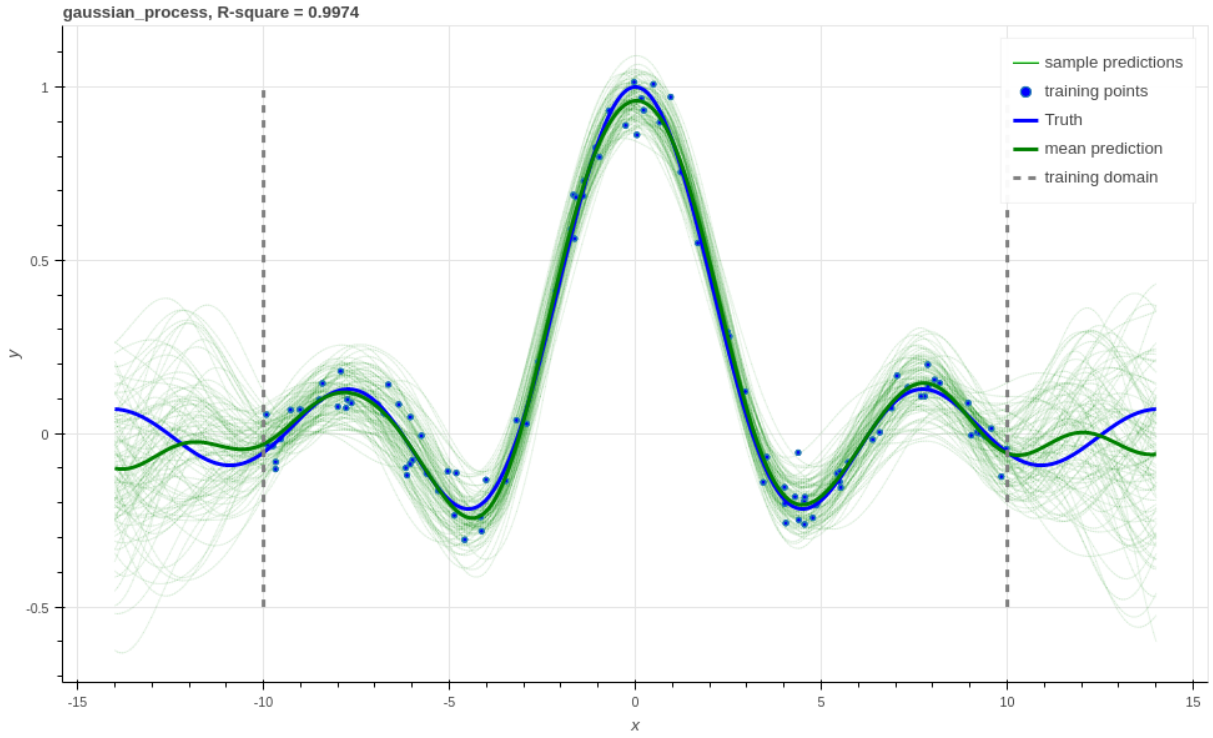


Fig. 7.12: Gaussian process regression, RBF kernel, R-square = 0.9974.

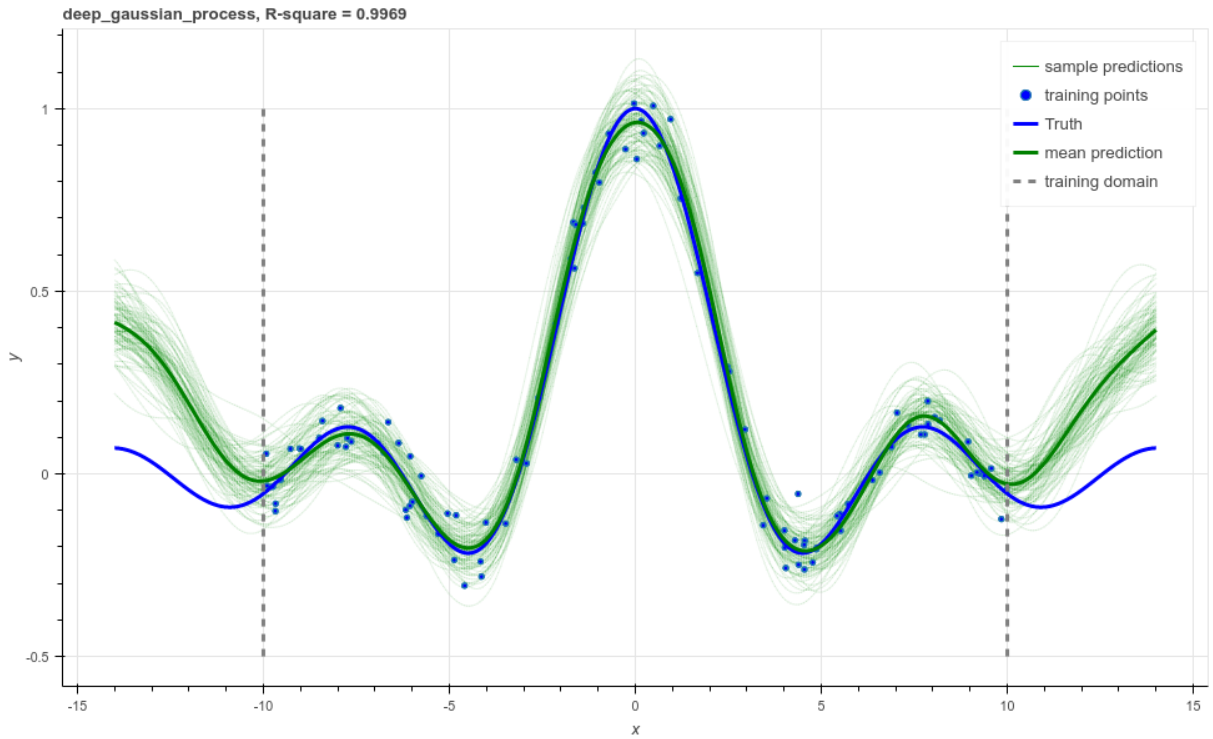


Fig. 7.13: Deep Gaussian process regression, RBF kernel, R-square = 0.9969.

And that is it! We hope this tutorial conveys just how flexible Aboleth is in allowing you to construct different models. You can find the code used to generate these figures and results in this tutorial with the demos here.

References

7.6.3 Integrating Aboleth with Keras

In most circumstances, Aboleth's layer composition framework is interoperable with TensorFlow and Keras layers. This gives us access to a vast range of layers not directly implemented in Aboleth which are suitable for various problems, such as LSTMs, GRUs and other variants of recurrent layers for sequence prediction, to name just one example. Furthermore, this also allows us to readily take advantage of the various sophisticated normalization and activation layers proposed in the current research, such as Batch Normalization[1], Leaky ReLU, ELU[2], et cetera.

Here we define a simple wrapper layer that allows one to plug in Keras / TensorFlow layers.

```
class WrapperLayer(SampleLayer):

    def __init__(self, layer, *args, **kwargs):

        self.layer = layer(*args, **kwargs)

    def _build(self, X):
        """Build the graph of this layer."""

        # keras first flattens the `n - len(input_shape)` dimensions
        # of the array and operates on last `len(input_shape)` dimensions
        # which has shape `input_shape`
        Net = self.layer(X)
        # aggregate layer regularization terms
        KL = tf.reduce_sum(self.layer.losses)

        return Net, KL
```

Now we can use the wrapped layers and Aboleth's native layers interchangeably. For example, consider tackling a toy regression problem using a deep neural net with dropout layers, where we perform maximum a posteriori (MAP) estimation of the layer weights / biases. The following are effectively equivalent:

```
net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    ab.DenseMAP(output_dim=64, l2_reg=0.01, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DropOut(keep_prob=.5) >>
    ab.DenseMAP(output_dim=64, l2_reg=0.01, l1_reg=0.) >>
    ab.Activation(tf.tanh) >>
    ab.DropOut(keep_prob=.5) >>
    ab.DenseMAP(output_dim=1, l2_reg=0.01, l1_reg=0.)
)
```

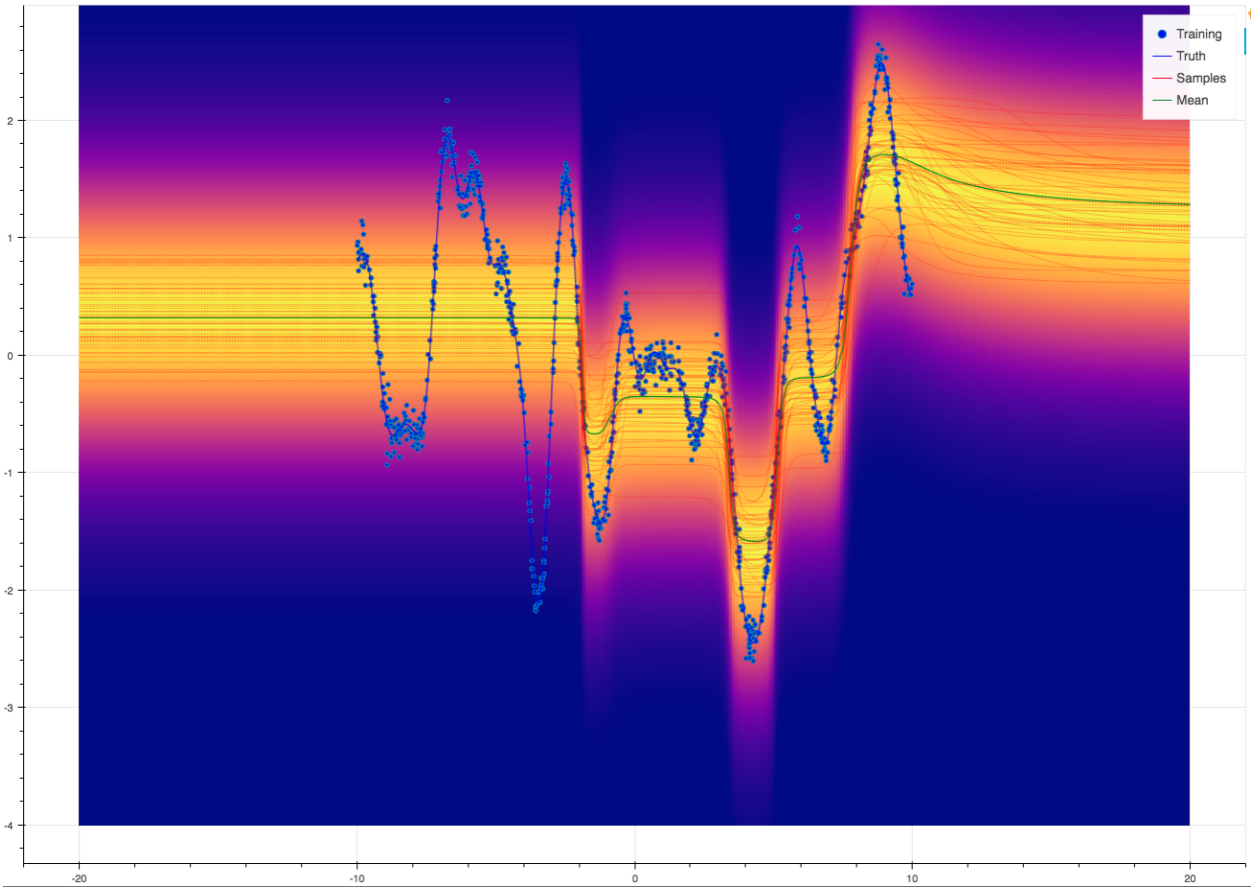
```
l1_l2_reg = tf.keras.regularizers.l1_l2(l1=0., l2=0.01)

net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    WrapperLayer(tf.keras.layers.Dense, units=64, activation='tanh',
```

[1] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in Proceedings of the 32nd International Conference on Machine Learning, 2015, vol. 37, pp. 448–456.

[2] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)," Nov. 2015.

```
                kernel_regularizer=l1_l2_reg, bias_regularizer=l1_l2_reg) >>
    ab.DropOut(keep_prob=.5) >>
    WrapperLayer(tf.keras.layers.Dense, units=64, activation='tanh',
                kernel_regularizer=l1_l2_reg, bias_regularizer=l1_l2_reg) >>
    ab.DropOut(keep_prob=.5) >>
    WrapperLayer(tf.keras.layers.Dense, units=1, kernel_regularizer=l1_l2_reg,
                bias_regularizer=l1_l2_reg)
)
```



Now it's easy to augment this model by incorporating other building blocks from Keras, e.g.

```
net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    WrapperLayer(tf.keras.layers.Dense, units=64) >>
    WrapperLayer(tf.keras.layers.BatchNormalization) >>
    WrapperLayer(tf.keras.layers.LeakyReLU) >>
    ab.DropOut(keep_prob=.5) >>

    WrapperLayer(tf.keras.layers.Dense, units=64) >>
    WrapperLayer(tf.keras.layers.BatchNormalization) >>
    WrapperLayer(tf.keras.layers.LeakyReLU) >>
    ab.DropOut(keep_prob=.5) >>

    WrapperLayer(tf.keras.layers.Dense, units=1, kernel_regularizer=l1_l2_reg,
                bias_regularizer=l1_l2_reg)
)
```

Or use it to perform classification on sequences:

```
net = (
    ab.InputLayer(name="X", n_samples=n_samples_) >>
    WrapperLayer(tf.keras.layers.LSTM, units=128) >>
    ab.DropOut(keep_prob=.5) >>
    WrapperLayer(tf.keras.layers.Dense, units=1)
)
```

You can find the script here: regression_keras.py

7.7 API

This is the application programming interface guide for Aboleth.

Aboleth is implemented in pure Python 3 only (we don't test Python 2, and we may use Python 3 language specific features). If you would like to contribute (please do), see the *Contributing Guidelines*.

7.7.1 ab.losses

Network loss functions.

aboleth.losses.**elbo**(*likelihood*, *Y*, *N*, *KL*, *like_weights=None*)

Build the evidence lower bound loss for a neural net.

Parameters

- **likelihood** (*tf.distributions.Distribution*) – the likelihood object that takes neural network(s) as an input. The `batch_shape` of this object should be `(n_samples, N, ...)`, where `n_samples` is the number of likelihood samples (defined by ab.InputLayer) and `N` is the number of observations (can be `?` if you are using a placeholder and mini-batching).
- **Y** (*ndarray, Tensor*) – the targets of shape `(N, tasks)`.
- **N** (*int, Tensor*) – the total size of the dataset (i.e. number of observations).
- **KL** (*float, Tensor*) – the Kullback Leibler divergence between the posterior and prior parameters of the model ($\mathrm{KL}[q||p]$).
- **like_weights** (*ndarray, Tensor*) – weights to apply to each observation in the expected log likelihood. This should be a tensor/array of shape `(N,)` (or a shape that prevents broadcasting).

Returns **nelbo** – the loss function of the Bayesian neural net (negative ELBO).

Return type Tensor

aboleth.losses.**max_posterior**(*likelihood*, *Y*, *regulariser*, *like_weights=None*)

Build maximum a-posteriori (MAP) loss for a neural net.

Parameters

- **likelihood** (*tf.distributions.Distribution*) – the likelihood object that takes neural network(s) as an input. The `batch_shape` of this object should be `(n_samples, N, ...)`, where `n_samples` is the number of likelihood samples (defined by ab.InputLayer) and `N` is the number of observations (can be `?` if you are using a placeholder and mini-batching).

- **Y** (*ndarray, Tensor*) – the targets of shape `(N, tasks)`.
- **regulariser** (*float, Tensor*) – the regulariser on the parameters of the model to penalise model complexity.
- **like_weights** (*ndarray, Tensor*) – weights to apply to each observation in the expected log likelihood. This should be a tensor/array of shape `(N,)` (or a shape that prevents broadcasting).

Returns **map** – the loss function of the MAP neural net.

Return type Tensor

7.7.2 ab.baselayers

Base Classes for Layers.

class aboleth.baselayers.**Layer**
Bases: `object`

Layer base class.

This is an identity layer, and is primarily meant to be subclassed to construct more intersting layers.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.baselayers.**LayerComposite**(**layers*)
Bases: *aboleth.baselayers.Layer*

Composition of Layers.

Parameters ***layers** – the layers to compose. All must be of type Layer.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.baselayers.**MultiLayer**
Bases: `object`

Base class for layers that take multiple inputs as kwargs.

This is an Abstract class as there is no canonical identity for this layer (because it must do some kind of reduction).

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.baselayers.**MultiLayerComposite**(**layers*)
Bases: *aboleth.baselayers.MultiLayer*

Composition of MultiLayers.

Parameters ***layers** – the layers to compose. First layer must be of type Multilayer, subsequent layers must be of type Layer.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

aboleth.baselayers.**stack**(*l*, **layers*)
Stack multiple Layers.

This is a convenience function that acts as an alternative to the rshift operator implemented for Layers and Multilayers. It is syntatically more compact for stacking large numbers of layers or lists of layers.

The type of stacking (Layer or Multilayer) is dispatched on the first argument.

Parameters

- **l** (*Layer or MultiLayer*) – The first layer to stack. The type of this layer determines the type of the output; MultiLayerComposite or LayerComposite.
- ***layers** – list of additional layers to stack. Must all be of type Layer, because function composition only works with the first function having multiple arguments.

Returns **result** – A single layer that is the composition of the input layers.

Return type *MultiLayerComposite* or *LayerComposite*

7.7.3 ab.layers

Network layers and utilities.

class aboleth.layers.**Activation**(*h=<function Activation.<lambda>>*)
Bases: *aboleth.baselayers.Layer*

Activation function layer.

Parameters **h** (*callable*) – the *element-wise* activation function.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**Conv2DMAP**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='SAME'*, *l1_reg=1.0*, *l2_reg=1.0*, *use_bias=True*)

Bases: *aboleth.layers.SampleLayer*

A 2D convolution layer, with maximum a posteriori (MAP) inference.

This layer uses maximum *a-posteriori* inference to learn the convolutional kernels and biases, and so also returns complexity penalities (l1 or l2) for the weights and biases.

Parameters

- **filters** (*int*) – the dimension of the output of this layer (i.e. the number of filters in the convolution).
- **kernel_size** (*int, tuple or list*) – width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides** (*int, tuple or list*) – the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions
- **padding** (*str*) – One of 'SAME' or 'VALID'. Defaults to 'SAME'. The type of padding algorithm to use.
- **l1_reg** (*float*) – the value of the l1 weight regularizer,

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**Conv2DVariational**(*filters*, *kernel_size*, *strides=(1, 1)*, *padding='SAME'*, *std=1.0*, *use_bias=True*, *prior_W=None*, *prior_b=None*, *post_W=None*, *post_b=None*)

Bases: *aboleth.layers.SampleLayer*

A 2D convolution layer, with variational inference.

(Does not currently support full covariance weights.)

Parameters

- **filters** (*int*) – the dimension of the output of this layer (i.e. the number of filters in the convolution).
- **kernel_size** (*int, tuple or list*) – width and height of the 2D convolution window. Can be a single integer to specify the same value for all spatial dimensions.
- **strides** (*int, tuple or list*) – the strides of the convolution along the height and width. Can be a single integer to specify the same value for all spatial dimensions
- **padding** (*str*) – One of 'SAME' or 'VALID'. Defaults to 'SAME'. The type of padding algorithm to use.
- **std** (*float*) – the initial value of the weight prior standard deviation (σ above), this is optimized a la maximum likelihood type II.
- **use_bias** (*bool*) – If true, also learn a bias weight, e.g. a constant offset weight.

- **prior_W** (*tf.distributions.Distribution, optional*) – This is the prior distribution object to use on the layer weights. It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `std` parameter.
- **prior_b** (*tf.distributions.Distribution, optional*) – This is the prior distribution object to use on the layer intercept. It must have parameters compatible with (output_dim,) shaped weights. This ignores the `std` and `use_bias` parameters.
- **post_W** (*tf.distributions.Distribution, optional*) – It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `full` parameter. See also `distributions.gaus_posterior`.
- **post_b** (*tf.distributions.Distributions, optional*) – This is the posterior distribution object to use on the layer intercept. It must have parameters compatible with (output_dim,) shaped weights. This ignores the `use_bias` parameters. See also `distributions.norm_posterior`.

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**DenseMAP**(*output_dim*, *l1_reg=1.0*, *l2_reg=1.0*, *use_bias=True*)

Bases: *aboleth.layers.SampleLayer*

Dense (fully connected) linear layer, with MAP inference.

This implements a linear layer, and when called returns

$$f(\mathbf{X}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

where $\mathbf{X} \in \mathbb{R}^{N \times D_{in}}$, $\mathbf{W} \in \mathbb{R}^{D_{in} \times D_{out}}$ and $\mathbf{b} \in \mathbb{R}^{D_{out}}$. This layer uses maximum *a-posteriori* inference to learn the weights and biases, and so also returns complexity penalities (l1 or l2) for the weights and biases.

Parameters

- **output_dim** (*int*) – the dimension of the output of this layer
- **l1_reg** (*float*) – the value of the l1 weight regularizer,

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**DenseVariational**(*output_dim*, *std=1.0*, *full=False*, *use_bias=True*, *prior_W=None*, *prior_b=None*, *post_W=None*, *post_b=None*)

Bases: *aboleth.layers.SampleLayer3*

A dense (fully connected) linear layer, with variational inference.

This implements a dense linear layer,

$$f(\mathbf{X}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

where prior, $p(\cdot)$, and approximate posterior, $q(\cdot)$ distributions are placed on the weights and *also* the biases. Here $\mathbf{X} \in \mathbb{R}^{N \times D_{in}}$, $\mathbf{W} \in \mathbb{R}^{D_{in} \times D_{out}}$ and $\mathbf{b} \in \mathbb{R}^{D_{out}}$. By default, the same Normal prior is placed on each of the layer weights and biases,

$$w_{ij} \sim \mathcal{N}(0, \sigma^2), \quad b_j \sim \mathcal{N}(0, \sigma^2),$$

and a different Normal posterior is learned for each of the layer weights and biases,

$$w_{ij} \sim \mathcal{N}(m_{ij}, c_{ij}), \quad b_j \sim \mathcal{N}(l_j, o_j).$$

We also have the option of placing full-covariance Gaussian posteriors on the input dimension of the weights,

$$\mathbf{w}_j \sim \mathcal{N}(\mathbf{m}_j, \mathbf{C}_j),$$

where $\mathbf{m}_j \in \mathbb{R}^{D_{in}}$ and $\mathbf{C}_j \in \mathbb{R}^{D_{in} \times D_{in}}$.

This layer will use variational inference to learn *all* of the non-zero *prior* and posterior parameters.

Whenever this layer is called, it will return the result,

$$f^{(s)}(\mathbf{X}) = \mathbf{X}\mathbf{W}^{(s)} + \mathbf{b}^{(s)}$$

with samples from the posteriors, $\mathbf{W}^{(s)} \sim q(\mathbf{W})$ and $\mathbf{b}^{(s)} \sim q(\mathbf{b})$. The number of samples, *s*, can be controlled by using the `n_samples` argument in an `InputLayer` used to feed the first layer of a model, or by tiling $\mathbf{X}$ on the first dimension. This layer also returns the result of $\mathrm{KL}[q\|p]$ for all parameters.

Parameters

- **output_dim** (*int*) – the dimension of the output of this layer
- **std** (*float*) – the initial value of the weight prior standard deviation (σ above), this is optimized a la maximum likelihood type II.
- **full** (*bool*) – If true, use a full covariance Gaussian posterior for *each* of the output weight columns, otherwise use an independent (diagonal) Normal posterior.
- **use_bias** (*bool*) – If true, also learn a bias weight, e.g. a constant offset weight.
- **prior_W** (*tf.distributions.Distribution, optional*) – This is the prior distribution object to use on the layer weights. It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `std` parameter.
- **prior_b** (*tf.distributions.Distribution, optional*) – This is the prior distribution object to use on the layer intercept. It must have parameters compatible with (output_dim,) shaped weights. This ignores the `std` and `use_bias` parameters.
- **post_W** (*tf.distributions.Distribution, optional*) – It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `full` parameter. See also `distributions.gaus_posterior`.
- **post_b** (*tf.distributions.Distributions, optional*) – This is the posterior distribution object to use on the layer intercept. It must have parameters compatible with (output_dim,) shaped weights. This ignores the `use_bias` parameters. See also `distributions.norm_posterior`.

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**DropOut**(*keep_prob*, *observation_axis=1*)

Bases: *aboleth.baselayers.Layer*

Dropout layer, Bernoulli probability of not setting an input to zero.

This is just a thin wrapper around tf.dropout

Parameters

- **keep_prob** (*float, Tensor*) – the probability of keeping an input. See tf.dropout.
- **observation_axis** (*int*) – The axis that indexes the observations (N). This will assume the obserations are on the *second* axis, i.e. (n_samples, N, ...). This is so we can repeat the dropout pattern over observations, which has the effect of dropping out weights consistently, thereby sampling the "latent function" of the layer.

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**EmbedMAP**(*output_dim*, *n_categories*, *l1_reg=1.0*, *l2_reg=1.0*)

Bases: *aboleth.layers.SampleLayer3*

Dense (fully connected) embedding layer, with MAP inference.

This layer works directly inputs of *K* category *indices* rather than one-hot representations, for efficiency. Each column of the input is embedded seperately, and the result concatenated along the last axis. It is a dense linear layer,

$$f(\mathbf{X}) = \mathbf{XW}$$

Here $\mathbf{X} \in \mathbb{N}_2^{N \times K}$ and $\mathbf{W} \in \mathbb{R}^{K \times D_{out}}$. Though in code we represent $\mathbf{X}$ as a vector of indices in $\mathbb{N}_K^{N \times 1}$. This layer uses maximum *a-posteriori* inference to learn the weights and so also returns complexity penalities (l1 or l2) for the weights.

Parameters

- **output_dim** (*int*) – the dimension of the output (embedding) of this layer
- **n_categories** (*int*) – the number of categories in the input variable
- **l1_reg** (*float*) – the value of the l1 weight regularizer,

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer

- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**EmbedVariational**(*output_dim*, *n_categories*, *std=1.0*, *full=False*, *prior_W=None*, *post_W=None*)

Bases: *aboleth.layers.DenseVariational*

Dense (fully connected) embedding layer, with variational inference.

This layer works directly inputs of *K* category *indices* rather than one-hot representations, for efficiency. Each column of the input is embedded seperately, and the result concatenated along the last axis. It is a dense linear layer,

$$f(\mathbf{X}) = \mathbf{X}\mathbf{W},$$

where prior, $p(\cdot)$, and approximate posterior, $q(\cdot)$ distributions are placed on the weights. Here $\mathbf{X} \in \mathbb{N}_2^{N \times K}$ and $\mathbf{W} \in \mathbb{R}^{K \times D_{out}}$. Though in code we represent $\mathbf{X}$ as a vector of indices in $\mathbb{N}_K^{N \times 1}$. By default, the same Normal prior is placed on each of the layer weights,

$$w_{ij} \sim \mathcal{N}(0, \sigma^2),$$

and a different Normal posterior is learned for each of the layer weights,

$$w_{ij} \sim \mathcal{N}(m_{ij}, c_{ij}).$$

We also have the option of placing full-covariance Gaussian posteriors on the input dimension of the weights,

$$\mathbf{w}_j \sim \mathcal{N}(\mathbf{m}_j, \mathbf{C}_j),$$

where $\mathbf{m}_j \in \mathbb{R}^K$ and $\mathbf{C}_j \in \mathbb{R}^{K \times K}$.

This layer will use variational inference to learn *all* of the non-zero *prior* and posterior parameters.

Whenever this layer is called, it will return the result,

$$f^{(s)}(\mathbf{X}) = \mathbf{X}\mathbf{W}^{(s)}$$

with samples from the posterior, $\mathbf{W}^{(s)} \sim q(\mathbf{W})$. The number of samples, *s*, can be controlled by using the `n_samples` argument in an `InputLayer` used to feed the first layer of a model, or by tiling $\mathbf{X}$ on the first dimension. This layer also returns the result of $\mathrm{KL}[q\|p]$ for all parameters.

Parameters

- **output_dim** (*int*) – the dimension of the output (embedding) of this layer
- **n_categories** (*int*) – the number of categories in the input variable
- **std** (*float*) – the initial value of the weight prior standard deviation (σ above), this is optimized a la maximum likelihood type II.
- **full** (*bool*) – If true, use a full covariance Gaussian posterior for *each* of the output weight columns, otherwise use an independent (diagonal) Normal posterior.
- **prior_W** (*tf.distributions.Distribution, optional*) – This is the prior distribution object to use on the layer weights. It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `std` parameter.
- **post_W** (*tf.distributions.Distribution, optional*) – This is the posterior distribution object to use on the layer weights. It must have parameters compatible with (input_dim, output_dim) shaped weights. This ignores the `full` parameter. See also `distributions.gaus_posterior`.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**InputLayer**(*name*, *n_samples=1*)
Bases: *aboleth.baselayers.MultiLayer*

Create an input layer.

This layer defines input kwargs so that a user may easily provide the right inputs to a complex set of layers. It takes a tensor of shape `(N, ...)`. The input is tiled along a new first axis creating a `(n_samples, N, ...)` tensor for propagating samples through a variational deep net.

Parameters

- **name** (*string*) – The name of the input. Used as the argument for input into the net.
- **n_samples** (*int, Tensor*) – The number of samples to propagate through the network. We recommend making this a `tf.placeholder` so you can vary it as required.

Note: We recommend making `n_samples` a `tf.placeholder` so it can be varied between training and prediction!

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**MaxPool2D**(*pool_size*, *strides*, *padding='SAME'*)
Bases: *aboleth.baselayers.Layer*

Max pooling layer for 2D inputs (e.g. images).

This is just a thin wrapper around tf.nn.max_pool

Parameters

- **pool_size** (*tuple or list of 2 ints*) – width and height of the pooling window.
- **strides** (*tuple or list of 2 ints*) – the strides of the pooling operation along the height and width.
- **padding** (*str*) – One of 'SAME' or 'VALID'. Defaults to 'SAME'. The type of padding

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**RandomArcCosine**(*n_features*, *lenscale=1.0*, *p=1*, *variational=False*, *lenscale_posterior=None*)

Bases: *aboleth.layers.RandomFourier*

Random arc-cosine kernel layer.

Parameters

- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.
- **lenscale** (*float, ndarray, Tensor*) – the lenght scales of the ar-cosine kernel, this can be a scalar for an isotropic kernel, or a vector for an automatic relevance detection (ARD) kernel.
- **p** (*int*) – The order of the arc-cosine kernel, this must be an integer greater than, or eual to zero. 0 will lead to sigmoid-like kernels, 1 will lead to relu-like kernels, 2 quadratic-relu kernels etc.
- **variational** (*bool*) – use variational features instead of random features, (i.e. VAR-FIXED in [2]).
- **lenscale_posterior** (*float, ndarray, optional*) – the *initial* value for the posterior length scale. This is only used if `variational==True`. This can be a scalar or vector (different initial value per input dimension). If this is left as None, it will be set to `sqrt(1 / input_dim)` (this is similar to the 'auto' setting for a scikit learn SVM with a RBF kernel).

Note: This should be followed by a dense layer to properly implement a kernel approximation.

See also:

[1] Cho, Youngmin, and Lawrence K. Saul. "Analysis and extension of arc-cosine kernels for large margin classification." arXiv preprint arXiv:1112.3712 (2011).

[2] Cutajar, K. Bonilla, E. Michiardi, P. Filippone, M. Random Feature Expansions for Deep Gaussian Processes. In ICML, 2017.

__call__(*X*)

Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**RandomFourier**(*n_features*, *kernel*)

Bases: *aboleth.layers.SampleLayer3*

Random Fourier feature (RFF) kernel approximation layer.

Parameters

- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.

- **kernel** (kernels.ShiftInvariant) – the kernel object that yeilds the random samples from the fourier spectrum of a particular kernel to approximate. See the *ab.kernels* module.

Note: This should be followed by a dense layer to properly implement a kernel approximation.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**Reshape**(*target_shape*)
Bases: *aboleth.baselayers.Layer*

Reshape layer.

Reshape and output an tensor to a specified shape.

Parameters **targe_shape** (*tuple of ints*) – Does not include the samples or batch axes.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**SampleLayer**
Bases: *aboleth.baselayers.Layer*

Sample Layer base class.

This is the base class for layers that build upon stochastic (variational) nets. These expect *rank >= 3* input Tensors, where the first dimension indexes the random samples of the stochastic net.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.layers.**SampleLayer3**
Bases: *aboleth.layers.SampleLayer*

Special case of SampleLayer restricted to *rank == 3* input Tensors.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

7.7.4 ab.hlayers

Higher-order neural network layers (made from other layers).

class aboleth.hlayers.**Concat**(**layers*)
Bases: *aboleth.baselayers.MultiLayer*

Concatenates the output of multiple layers.

Parameters **layers** (*[MultiLayer]*) – The layers to concatenate.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.hlayers.**PerFeature**(**layers*, *slices=None*)
Bases: *aboleth.baselayers.Layer*

Concatenate multiple layers with sliced inputs.

Each layer will recieve a slice along the last axis of the input to the new function. In other words, `PerFeature(l1, l2)(X)` will call `l1(X[..., 0]) and l2(X[..., 1])` then concatenate their outputs into a single tensor. This is mostly useful for simplifying embedding multiple categorical inputs that are stored columnwise in the same 2D tensor.

This function assumes the tensor being provided is 3D.

Parameters

- **layers** (*[Layer]*) – The layers to concatenate.
- **slices** (*[slice]*) – The slices into X to give to each layer, this has to be the same length as layers. If this is None, it will give *columns* of X to each layer, the number of columns is determined by the number of layers.

__call__(*X*)
Construct the subgraph for this layer.

Parameters **X** (*Tensor*) – the input to this layer

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.hlayers.**Sum**(**layers*)
Bases: *aboleth.baselayers.MultiLayer*

Sums multiple layers by adding their outputs.

Parameters **layers** (*[MultiLayer]*) – The layers to add.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

7.7.5 ab.kernels

Random kernel classes for use with the RandomKernel layers.

class aboleth.kernels.**Matern**(*lenscale=1.0*, *p=1*)
Bases: *aboleth.kernels.ShiftInvariant*

Matern kernel approximation.

Parameters

- **lenscale** (*float, ndarray, Tensor, Variable*) – the length scales of the shift invariant kernel, this can be a scalar for an isotropic kernel, or a vector of shape (input_dim, 1) for an automatic relevance detection (ARD) kernel. If you wish to learn this parameter, make it a Variable (or `ab.pos(tf.Variable(...))` to keep it positively constrained).
- **p** (*int*) – a zero or positive integer specifying the number of the Matern kernel, e.g. `p == 0` results int a Matern 1/2 kernel, `p == 1` results in the Matern 3/2 kernel etc.

weights(*input_dim*, *n_features*, *dtype=<class 'numpy.float32'>*)
Generate the random fourier weights for this kernel.

Parameters

- **input_dim** (*int*) – the input dimension to this layer.
- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.
- **dtype** (*np.dtype*) – the dtype of the features to draw, this should match the observations.

Returns

- **P** (*ndarray*) – the random weights of the fourier features of shape `(input_dim, n_features)`.
- **KL** (*Tensor, float*) – the KL penalty associated with the parameters in this kernel (0.0).

class aboleth.kernels.**RBF**(*lenscale=1.0*)
Bases: *aboleth.kernels.ShiftInvariant*

Radial basis kernel approximation.

Parameters **lenscale** (*float, ndarray, Tensor, Variable*) – the length scales of the shift invariant kernel, this can be a scalar for an isotropic kernel, or a vector of shape (input_dim, 1) for an automatic relevance detection (ARD) kernel. If you wish to learn this parameter, make it a Variable (or `ab.pos(tf.Variable(...))` to keep it positively constrained).

weights(*input_dim*, *n_features*, *dtype=<class 'numpy.float32'>*)
Generate the random fourier weights for this kernel.

Parameters

- **input_dim** (*int*) – the input dimension to this layer.
- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.
- **dtype** (*np.dtype*) – the dtype of the features to draw, this should match the observations.

Returns

- **P** (*ndarray*) – the random weights of the fourier features of shape `(input_dim, n_features)`.
- **KL** (*Tensor, float*) – the KL penalty associated with the parameters in this kernel (0.0).

class aboleth.kernels.**RBFVariational**(*lenscale=1.0*, *lenscale_posterior=None*)

Bases: *aboleth.kernels.ShiftInvariant*

Variational Radial basis kernel approximation.

This kernel is similar to the RBF kernel, however we learn an independant Gaussian posterior distribution over the kernel weights to sample from.

Parameters

- **lenscale** (*float, ndarray, Tensor, Variable*) – the length scales of the shift invariant kernel, this can be a scalar for an isotropic kernel, or a vector of shape (input_dim, 1) for an automatic relevance detection (ARD) kernel. If you wish to learn this parameter, make it a Variable (or `ab.pos(tf.Variable(...))` to keep it positively constrained).
- **lenscale_posterior** (*float, ndarray, optional*) – the *initial* value for the posterior length scale, this can be a scalar or vector (different initial value per input dimension). If this is left as None, it will be set to `sqrt(1 / input_dim)` (this is similar to the 'auto' setting for a scikit learn SVM with a RBF kernel).

weights(*input_dim*, *n_features*, *dtype=<class 'numpy.float32'>*)

Generate the random fourier weights for this kernel.

Parameters

- **input_dim** (*int*) – the input dimension to this layer.
- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.
- **dtype** (*np.dtype*) – the dtype of the features to draw, this should match the observations.

Returns

- **P** (*ndarray*) – the random weights of the fourier features of shape `(input_dim, n_features)`.
- **KL** (*Tensor, float*) – the KL penalty associated with the parameters in this kernel.

class aboleth.kernels.**ShiftInvariant**(*lenscale=1.0*)

Bases: `object`

Abstract base class for shift invariant kernel approximations.

Parameters **lenscale** (*float, ndarray, Tensor, Variable*) – the length scales of the shift invariant kernel, this can be a scalar for an isotropic kernel, or a vector of shape (input_dim, 1) for an automatic relevance detection (ARD) kernel. If you wish to learn this parameter, make it a Variable (or `ab.pos(tf.Variable(...))` to keep it positively constrained).

weights(*input_dim*, *n_features*, *dtype=<class 'numpy.float32'>*)

Generate the random fourier weights for this kernel.

Parameters

- **input_dim** (*int*) – the input dimension to this layer.
- **n_features** (*int*) – the number of unique random features, the actual output dimension of this layer will be `2 * n_features`.
- **dtype** (*np.dtype*) – the dtype of the features to draw, this should match the observations.

Returns

- **P** (*ndarray*) – the random weights of the fourier features of shape `(input_dim, n_features)`.
- **KL** (*Tensor, float*) – the KL penalty associated with the parameters in this kernel.

7.7.6 ab.distributions

Helper functions for model parameter distributions.

aboleth.distributions.**gaus_posterior**(*dim*, *std0*)

Initialise a posterior Gaussian distribution with a diagonal covariance.

Even though this is initialised with a diagonal covariance, a full covariance will be learned, using a lower triangular Cholesky parameterisation.

Parameters

- **dim** (*tuple or list*) – the dimension of this distribution.
- **std0** (*float*) – the initial (unoptimized) diagonal standard deviation of this distribution.

Returns **Q** – the initialised posterior Gaussian object.

Return type tf.contrib.distributions.MultivariateNormalTriL

Note: This will make tf.Variables on the randomly initialised mean and covariance of the posterior. The initialisation of the mean is from a Normal with zero mean, and `std0` standard deviation, and the initialisation of the (lower triangular of the) covariance is from a gamma distribution with an alpha of `std0` and a beta of 1.

---

aboleth.distributions.**kl_sum**(*q*, *p*)

Compute the total KL between (potentially) many distributions.

I.e. $\sum_i \mathrm{KL}[q_i||p_i]$

Parameters

- **q** (*tf.distributions.Distribution*) – A tensorflow Distribution object
- **p** (*tf.distributions.Distribution*) – A tensorflow Distribution object

Returns **kl** – the result of the sum of the KL divergences of the `q` and `p` distibutions.

Return type Tensor

aboleth.distributions.**norm_posterior**(*dim*, *std0*)
Initialise a posterior (diagonal) Normal distribution.

Parameters

- **dim** (*tuple or list*) – the dimension of this distribution.
- **std0** (*float*) – the initial (unoptimized) standard deviation of this distribution.

Returns **Q** – the initialised posterior Normal object.

Return type tf.distributions.Normal

Note: This will make tf.Variables on the randomly initialised mean and standard deviation of the posterior. The initialisation of the mean is from a Normal with zero mean, and `std0` standard deviation, and the initialisation of the standard deviation is from a gamma distribution with an alpha of `std0` and a beta of 1.

---

aboleth.distributions.**norm_prior**(*dim*, *std*)
Initialise a prior (zero mean, isotropic) Normal distribution.

Parameters

- **dim** (*tuple or list*) – the dimension of this distribution.
- **std** (*float*) – the prior standard deviation of this distribution.

Returns **P** – the initialised prior Normal object.

Return type tf.distributions.Normal

Note: This will make a tf.Variable on the variance of the prior that is initialised with `std`.

7.7.7 ab.impute

Layers that impute missing data.

class aboleth.impute.**FixedNormalImpute**(*datalayer*, *masklayer*, *mu_array*, *std_array*)
Bases: *aboleth.impute.ImputeOp*

Impute the missing values using marginal Gaussians over each column.

Takes two layers, one the returns a data tensor and the other returns a mask layer. Creates a layer that returns a tensor in which the masked values have been imputed as random draws from the marginal Gaussians.

Parameters

- **datalayer** (*callable*) – A layer that returns a data tensor. Must be of form `f(**kwargs)`.
- **masklayer** (*callable*) – A layer that returns a boolean mask tensor where True values are masked. Must be of form `f(**kwargs)`.
- **mu_array** (*array-like*) – A list of the global mean values of each dat column
- **std_array** (*array-like*) – A list of the global standard deviation of each data column

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.impute.**ImputeOp**(*datalayer*, *masklayer*)
Bases: *aboleth.baselayers.MultiLayer*

Abstract Base Impute operation. These specialise MultiLayers.

They expect a data InputLayer and a mask InputLayer. They return layers in which the masked values have been imputed.

Parameters

- **datalayer** (*callable*) – A layer that returns a data tensor. Must be of form `f(**kwargs)`.
- **masklayer** (*callable*) – A layer that returns a boolean mask tensor where True values are masked. Must be of form `f(**kwargs)`.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.impute.**LearnedNormalImpute**(*datalayer*, *masklayer*)
Bases: *aboleth.impute.ImputeOp*

Impute the missing values with draws from learned normal distributions.

Takes two layers, one the returns a data tensor and the other returns a mask layer. This creates a layer that will learn marginal Gaussian parameters per column, and infill missing values using draws from these Gaussians.

Parameters

- **datalayer** (*callable*) – A layer that returns a data tensor. Must be an InputLayer.
- **masklayer** (*callable*) – A layer that returns a boolean mask tensor where True values are masked. Must be an InputLayer.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.impute.**LearnedScalarImpute**(*datalayer*, *masklayer*)
Bases: *aboleth.impute.ImputeOp*

Impute the missing values using learnt scalar for each column.

Takes two layers, one the returns a data tensor and the other returns a mask layer. Creates a layer that returns a tensor in which the masked values have been imputed with a learned scalar value per colum.

Parameters

- **datalayer** (*callable*) – A layer that returns a data tensor. Must be an InputLayer.
- **masklayer** (*callable*) – A layer that returns a boolean mask tensor where True values are masked. Must be an InputLayer.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.impute.**MaskInputLayer**(*name*)
Bases: *aboleth.baselayers.MultiLayer*

Create an input layer for a binary mask tensor.

This layer defines input kwargs so that a user may easily provide the right binary mask inputs to a complex set of layers to enable imputation.

Parameters **name** (*string*) – The name of the input. Used as the agument for input into the net.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

class aboleth.impute.**MeanImpute**(*datalayer*, *masklayer*)
Bases: *aboleth.impute.ImputeOp*

Impute the missing values using the stochastic mean of their column.

Takes two layers, one the returns a data tensor and the other returns a mask layer. Returns a layer that returns a tensor in which the masked values have been imputed as the column means calculated from the batch.

Parameters

- **datalayer** (*callable*) – A layer that returns a data tensor. Must be of form `f(**kwargs)`.
- **masklayer** (*callable*) – A layer that returns a boolean mask tensor where True values are masked. Must be of form `f(**kwargs)`.

__call__(***kwargs*)
Construct the subgraph for this layer.

Parameters ****kwargs** – the inputs to this layer (Tensors)

Returns

- **Net** (*Tensor*) – the output of this layer
- **KL** (*float, Tensor*) – the regularizer/Kullback Leibler 'cost' of the parameters in this layer.

7.7.8 ab.random

Random generators and state.

class aboleth.random.**SeedGenerator**
Bases: object

Make new random seeds deterministically from a base random seed.

next()
Generate a random int using this object's base state.

Returns **result** – an integer that can be used to seed other random states deterministically.

Return type int

set_hyperseed(*hs*)
Set the random seed state in this object.

Parameters **hs** (*None, int, array_like*) – seed the random state of this object, see numpy.random.RandomState for valid inputs.

aboleth.random.**endless_permutations**(*N*)
Generate an endless sequence of permutations of the set [0, ..., N).

If we call this N times, we will sweep through the entire set without replacement, on the (N+1)th call a new permutation will be created, etc.

Parameters **N** (*int*) – the length of the set

Yields *int* – yeilds a random int from the set [0, ..., N)

Examples

```
>>> perm = endless_permutations(5)
>>> type(perm)
<class 'generator'>
>>> p = next(perm)
>>> p < 5
True
>>> p2 = next(perm)
>>> p2 != p
True
```

aboleth.random.**set_hyperseed**(*hs*)
Set the global hyperseed from which to generate all other seeds.

Parameters **hs** (*None, int, array_like*) – seed the random state of the global hyperseed, see numpy.random.RandomState for valid inputs.

7.7.9 ab.util

Package helper utilities.

aboleth.util.**batch**(*feed_dict*, *batch_size*, *n_iter=10000*, *N_=None*)
Create random batches for Stochastic gradients.

Feed dict data generator for SGD that will yeild random batches for a a defined number of iterations, which can be infinite. This generator makes consecutive passes through the data, drawing without replacement on each pass.

Parameters

- **feed_dict** (*dict of ndarrays*) – The data with `{tf.placeholder: data}` entries. This assumes all items have the *same* length!
- **batch_size** (*int*) – number of data points in each batch.
- **n_iter** (*int, optional*) – The number of iterations
- **N** (*tf.placeholder (int), optional*) – Place holder for the size of the dataset. This will be fed to an algorithm.

Yields *dict* – with each element an array length `batch_size`, i.e. a subset of data, and an element for `N_`. Use this as your feed-dict when evaluating a loss, training, etc.

`aboleth.util.`**batch_prediction**(*feed_dict*, *batch_size*)

Split the data in a feed_dict into contiguous batches for prediction.

Parameters

- **feed_dict** (*dict of ndarrays*) – The data with `{tf.placeholder: data}` entries. This assumes all items have the *same* length!
- **batch_size** (*int*) – number of data points in each batch.

Yields

- *ndarray* – an array of shape approximately (`batch_size`,) of indices into the original data for the current batch
- *dict* – with each element an array length `batch_size`, i.e. a subset of data. Use this as your feed-dict when evaluating a model, prediction, etc.

Note: The exact size of the batch may not be `batch_size`, but the nearest size that splits the size of the data most evenly.

`aboleth.util.`**pos**(*X*, *minval=1e-15*)

Constrain a `tf.Variable` to be positive only.

At the moment this is implemented as:

$$\max(|\mathbf{X}|, \text{minval})$$

This is fast and does not result in vanishing gradients, but will lead to non-smooth gradients and more local minima. In practice we haven't noticed this being a problem.

Parameters

- **X** (*Tensor*) – any Tensor in which all elements will be made positive.
- **minval** (*float*) – the minimum "positive" value the resulting tensor will have.

Returns **X** – a tensor the same shape as the input `X` but positively constrained.

Return type Tensor

Examples

```
>>> X = tf.constant(np.array([1.0, -1.0, 0.0]))
>>> Xp = pos(X)
>>> with tf.Session():
...     xp = Xp.eval()
```

```
>>> xp
array([  1.00000000e+00,   1.00000000e+00,   1.00000000e-15])
```

7.7.10 ab.datasets

CHAPTER 8

Feedback

If you have any suggestions or questions about **Aboleth** feel free to email us at lachlan.mccalman@data61.csiro.au or daniel.steinberg@data61.csiro.au.

If you encounter any errors or problems with **Aboleth**, please let us know! Open an Issue at the GitHub http://github.com/determinant-io/aboleth main repository.

Python Module Index

a

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

P

R

S

W