Aboleth Documentation

Release 0.6.0

Data61

Sep 27, 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.

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.
layers = (
    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. 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

CHAPTER 7

Documentation Contents

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.

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.

Logistic Classification

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

```python
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 inner-most 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,

```python
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):

```python
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:

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

Where:

```python
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(net, Y_, reg, likelihood)
```

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* module for a more comprehensive set of utilities (we recommend looking at tf.train.MonitoredTrainingSession, tf.train.limit_epochs and tf.train.shuffle_batch).

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})
```

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!

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,

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

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

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` 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 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 $s = \{1 \ldots 5\}$. These 5 samples are then passed to the `Activation` layer.

Then like before to complete the model specification:

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

```
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, `probs`,

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

now has a shape of $(5, N^*, 1)$ where we have 5 samples of N^* predictions; before we had $(N^*, 1)$. You can simply take the mean of these samples for the predicted class probability,

```
expected_p = np.mean(probs, axis=0)
```

or, you can generate *more* samples to get a more accurate expected probabilities (again with the TensorFlow session, `sess`),

```
probabilities = ab.predict_samples(net, feed_dict={X_: X_query},
                                   n_groups=10, session=sess)
```

This effectively calls `net` 10 times (`n_groups`) and concatenates the results into 50 samples (`n_groups * n_samples`), then we can take the mean of these samples exactly as before.

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,

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

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))
```

[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.

```
layers = (
    ab.InputLayer(name="X", n_samples=5) >>
    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);

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.

References

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.

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.contrib.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:

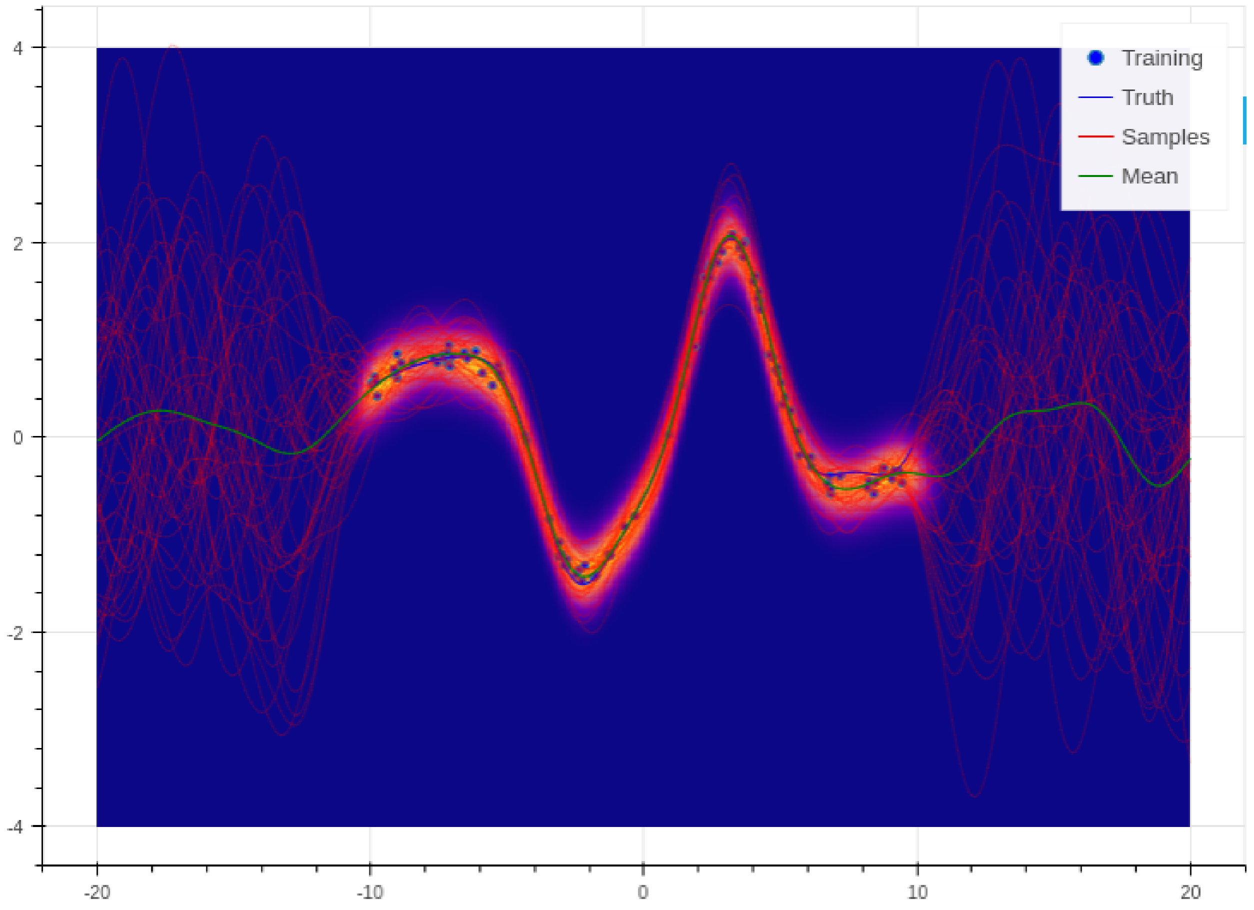
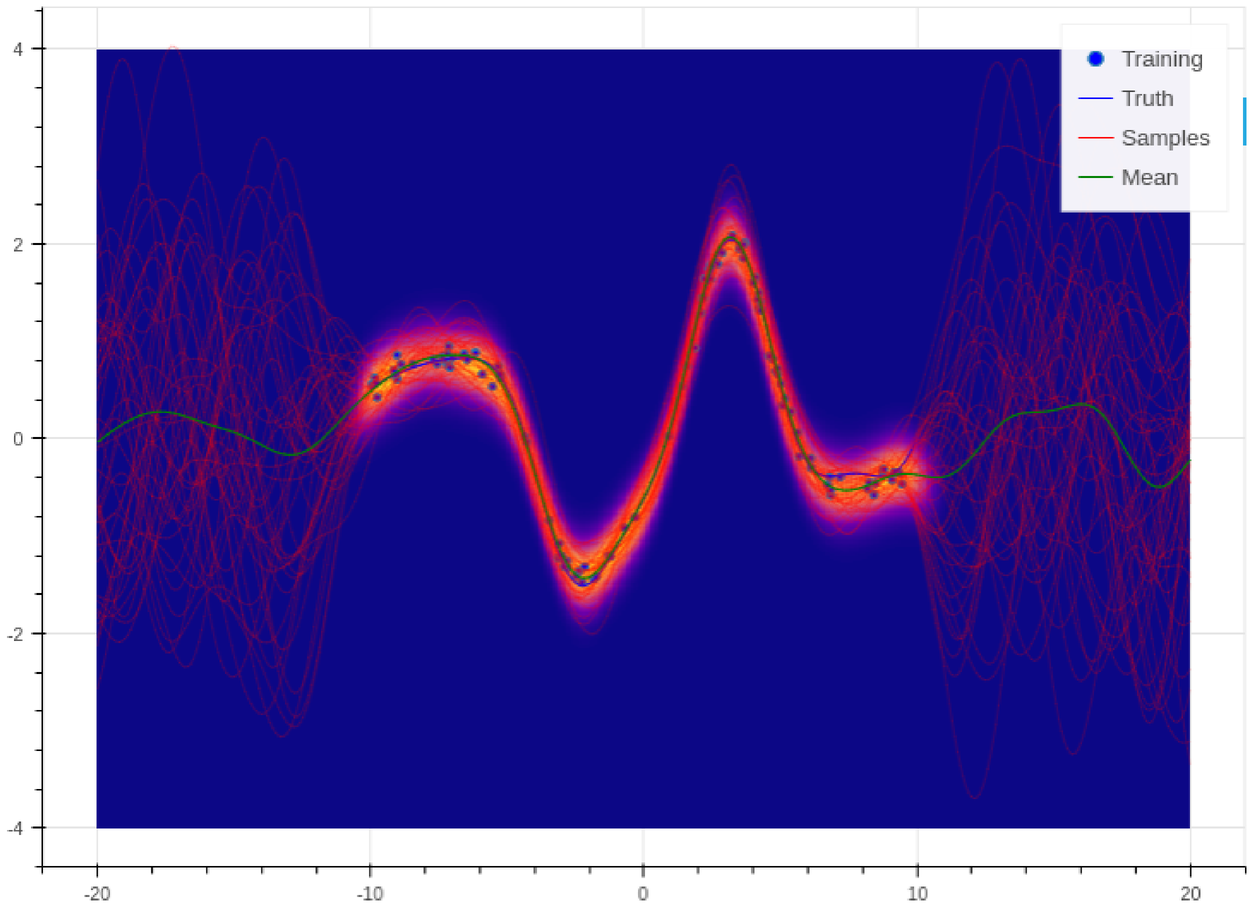


Fig. 7.1: Example of an approximate Gaussian process with a radial basis kernel. We have shown 50 samples of the predicted latent functions, the mean of these draws, and the heatmap is the probability of observing a target under the predictive distribution, $p(y^*|\mathbf{X}, \mathbf{y}, \mathbf{x}^*)$.


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

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

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*!!

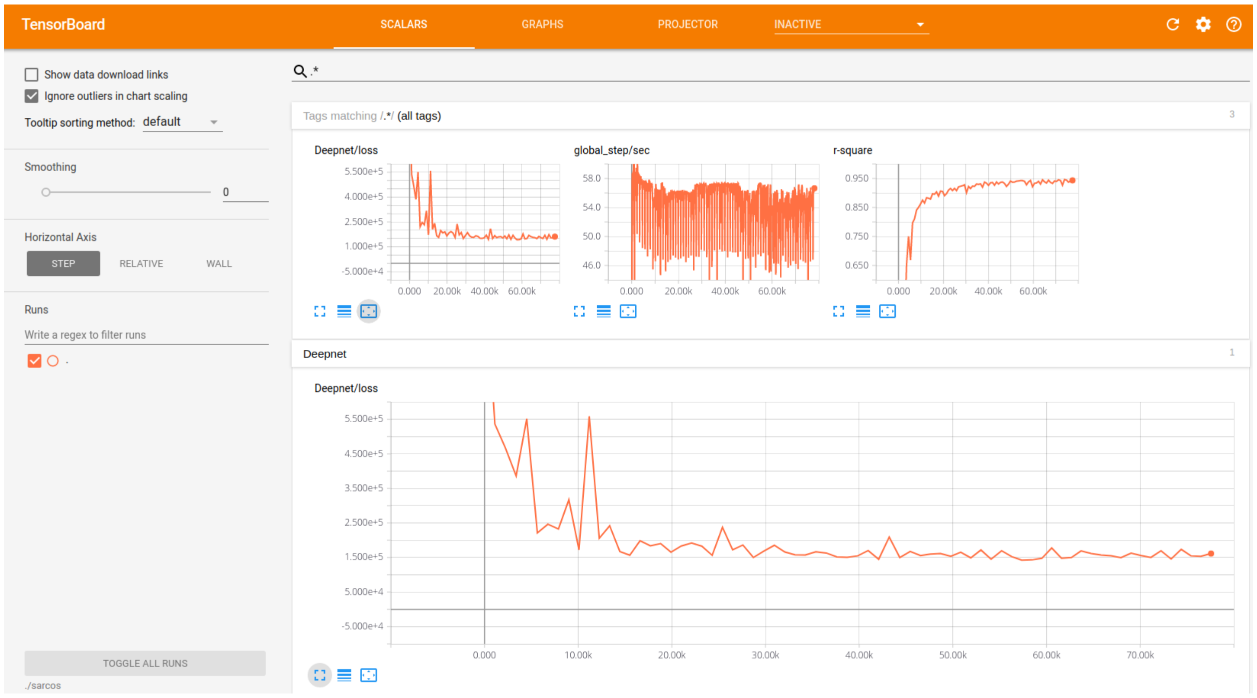


Fig. 7.2: Using TensorBoard to visualise the learning progress of the Aboleth model fitting the SARCOS dataset. The "r-square" plot here is made from evaluating the R-square performance on the held-out test set every epoch of training.

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.

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.

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

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

Authors

Development Leads

- Daniel Steinberg
- Lachlan McCalman
- Louis Tiao

Contributors

- Simon O'Callaghan
- Alistair Reid
- Joyce Wang

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.

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.

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
```

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.

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*.

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).
- **like_weights** (*callable, ndarray, Tensor*) – weights to apply to each observation in the expected log likelihood. This should be an array of shape (N,) or can be called as `like_weights(Y)` and should return a (N,) array.

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*, *first_axis_is_obs=True*)

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).
- **`like_weights`** (*`callable, ndarray, Tensor`*) – weights to apply to each observation in the expected log likelihood. This should be an array of shape (N,) or can be called as `like_weights(Y)` and should return a (N,) array.
- **`first_axis_is_obs`** (*`bool`*) – indicates if the first axis indexes the observations/data or not. This will be True if the likelihood outputs a `batch_shape` of (N, tasks) or False if `batch_shape` is (n_samples, N, tasks).

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

Return type Tensor

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*

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.`**`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 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*)

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.

__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 on shape (N, 1) inputs of *K* category *indices* rather than one-hot representations, for efficiency, and 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=None*)
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 2D tensor of shape (N, D). If n_samples is specified, the input is tiled along a new first axis creating a (n_samples, N, D) tensor for propogating samples through a variational deep net.

Parameters

- **name** (*string*) – The name of the input. Used as the agument for input into the net.
- **n_samples** (*int > 0*) – The number of samples.

__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.

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

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).

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.

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

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.

`__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.

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*`)`

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.

`__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.

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*)

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`.

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*)

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`.

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*)

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`.

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*)

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`.

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.

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`.

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.`**`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.

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.

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])
```

`aboleth.util.`**`predict_expected`**(*predictor*, *feed_dict=None*, *n_groups=1*, *session=None*)

Help to get the expected value from a predictor.

Parameters

- **predictor** (*Tensor*) – a tensor that outputs a shape (n_samples, N, tasks) where `n_samples` are the random samples from the predictor (e.g. the output of `Net`), `N` is the size of the query dataset, and `tasks` the number of prediction tasks.
- **feed_dict** (*dict, optional*) – The data with `{tf.placeholder: data}` entries.
- **n_groups** (*int*) – The number of times to evaluate the `predictor` and concatenate the samples.
- **session** (*Session*) – the session to be used to evaluate the predictor.

Returns **pred** – expected value of the prediction with shape (N, tasks). `n_samples * n_groups` samples go into evaluating this expectation.

Return type ndarray

Note: This has to be called in an *active* tensorflow session!

`aboleth.util.`**`predict_samples`**(*predictor*, *feed_dict=None*, *n_groups=1*, *session=None*)

Help to get samples from a predictor.

Parameters

- **`predictor`** (*`Tensor`*) – a tensor that outputs a shape (n_samples, N, tasks) where `n_samples` are the random samples from the predictor (e.g. the output of `Net`), `N` is the size of the query dataset, and `tasks` the number of prediction tasks.
- **`feed_dict`** (*`dict, optional`*) – The data with `{tf.placeholder: data}` entries.
- **`n_groups`** (*`int`*) – The number of times to evaluate the `predictor` and concatenate the samples.
- **`session`** (*`Session`*) – the session to be used to evaluate the predictor.

Returns **pred** – prediction samples of shape (n_samples * n_groups, N, tasks).

Return type ndarray

Note: This has to be called in an *active* tensorflow session!

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

Index