Aboleth

Bayesian Neural Networks and More

Dan Steinberg | Inference Systems Engineering

April 11, 2018

www.csiro.au

Outline

1. Background: Supervised learning to Bayesian neural networks
2. Aboleth:
    - Why another NN framework?
    - How its interface compares to other frameworks
    - Some of the internals
    - Examples

Supervised Learning to Bayesian NNs

Supervised learning

- $\mathbf{x}$ is a vector of covariates or features, y a target

Supervised learning

- $\mathbf{x}$ is a vector of covariates or features, y a target
- There exists an unknown function, f, that maps the covariates to the targets with some error,

$$y = f(\mathbf{x}) + \epsilon$$

Supervised learning

- **x** is a vector of covariates or features, y a target
- There exists an unknown function, f, that maps the covariates to the targets with some error,

$$y = f(\mathbf{x}) + \epsilon$$

- E.g. Houses have attributes, **x**, and market forces, f, control prices, y.

Supervised learning

- $\mathbf{x}$ is a vector of covariates or features, y a target
- There exists an unknown function, f, that maps the covariates to the targets with some error,

$$y = f(\mathbf{x}) + \epsilon$$

- *Supervised ML*: learn an approximation of this function, h, using examples,

$$y_i \approx h(\mathbf{x}_i) \quad \text{for all} \quad \{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$$

Supervised learning

- $\mathbf{x}$ is a vector of covariates or features, y a target
- There exists an unknown function, f, that maps the covariates to the targets with some error,

$$y = f(\mathbf{x}) + \epsilon$$

- *Supervised ML*: learn an approximation of this function, h, using examples,

$$y_i \approx h(\mathbf{x}_i) \quad \text{for all} \quad \{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$$

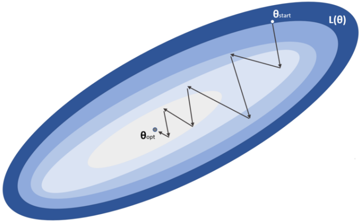
- E.g. Historical house sales data.

Supervised learning

- Usually we choose a class of h with parameters, θ,
- learning is **optimisation** of these parameters.

For example, find the θ that minimises the sum of squared errors

$$\hat{\theta} = \underset{\theta}{\text{argmin}} \frac{1}{N} \sum_{i=1}^{N} (y_i - h(\mathbf{x}_i, \theta))^2$$



Prediction

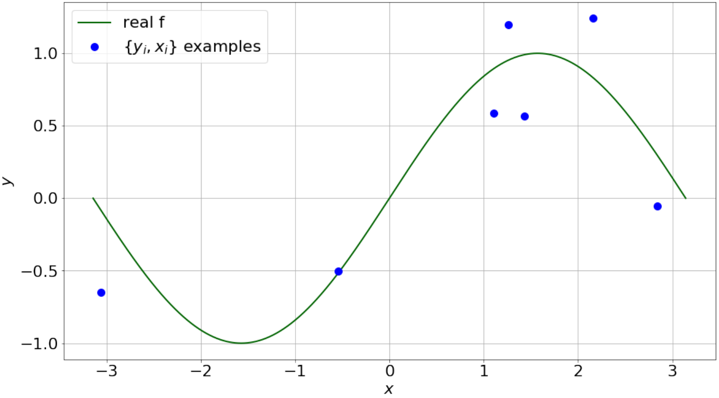
Once we have learned h, we want to use it to **predict** y^* for **new**, **unseen** instances of $\mathbf{x}^*$,

$$\mathbb{E}[y^*] = h(\mathbf{x}^*, \hat{\theta}).$$

Regression example

True function $f(x) = \text{sin}(x)$

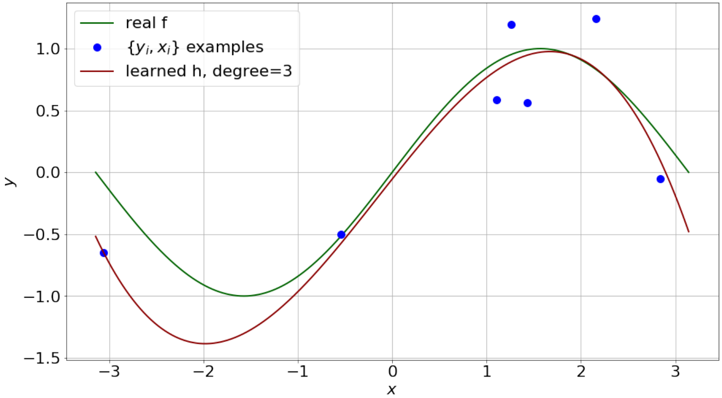
Observations $y_i = f(x_i) + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, 0.1)$



Fit a degree-3 polynomial

Model $h(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$

Objective $\text{argmin}_{\mathbf{w}} \frac{1}{N} \sum_{i=1}^{N} (y_i - h(x_i, \mathbf{w}))^2$



Linear Models

So we saw a polynomial is one class of model, e.g.:

$$
\begin{aligned}
h(x, \mathbf{w}) &= w_0 + w_1 x + w_2 x^2 + w_3 x^3 \\
&= \begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \\ x^3 \end{bmatrix} \\
&= \mathbf{w}^\top \mathrm{Poly}_3(x)
\end{aligned}
$$

This is part of a general class of models we call **linear** models,

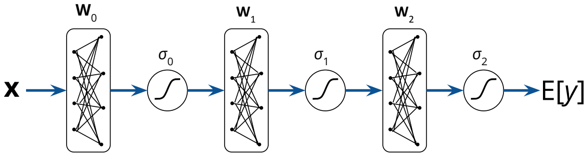
$$h(\mathbf{x}, \mathbf{w}) = \mathbf{w}^\top \phi(\mathbf{x})$$

where ϕ is some function.

Neural Nets

Neural nets generalise this class of linear models with **non-linearities** (σ) and **function composition**:

- 0 hidden layers: $\text{NN}_0(\mathbf{x}) = \sigma_0(\mathbf{W}_0\mathbf{x})$
- 1 hidden layer: $\text{NN}_1(\mathbf{x}) = \sigma_1(\mathbf{W}_1\sigma_0(\mathbf{W}_0\mathbf{x}))$
- L hidden layers: $\text{NN}_L(\mathbf{x}) = \sigma_L(\mathbf{W}_L\sigma_{L-1}(\mathbf{W}_{L-1}\ldots\sigma(\mathbf{W}_0\mathbf{x})))$

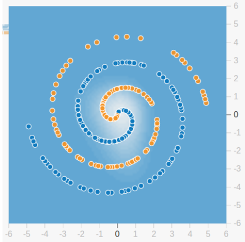


Why this representation?

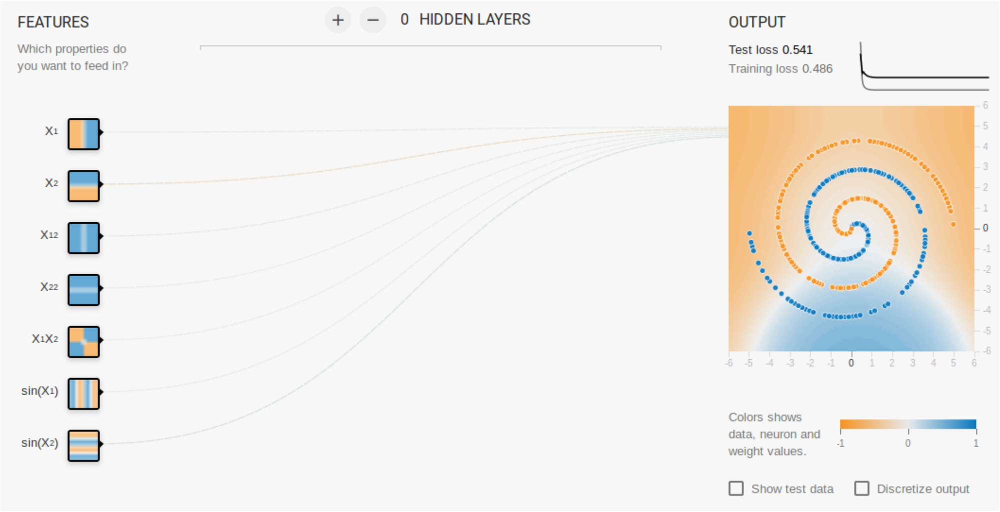
Neural nets can represent any function! But ...

- The more complex the function, the 'wider' the layers have to be (for a given depth).
- Or we can use (exponentially) 'narrower' and **deeper** nets!

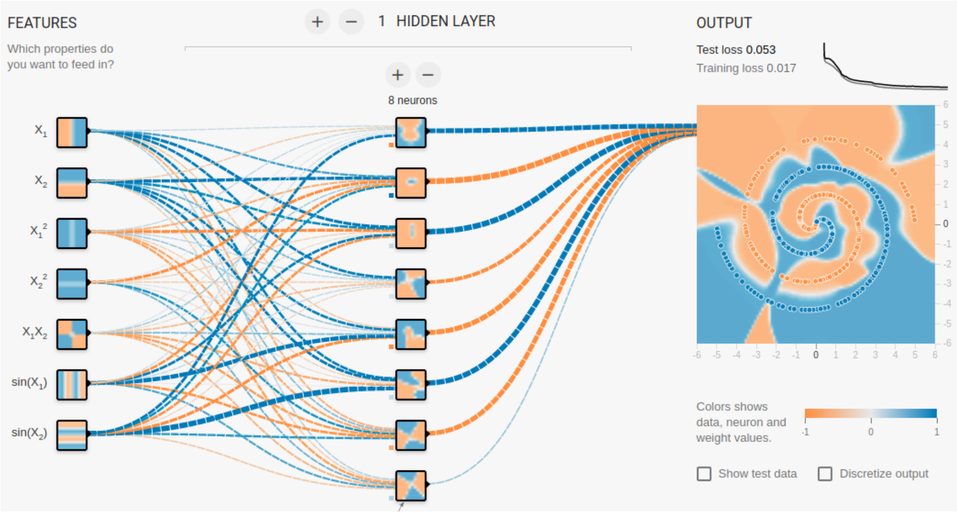
Nice demo: TensorFlow Playground



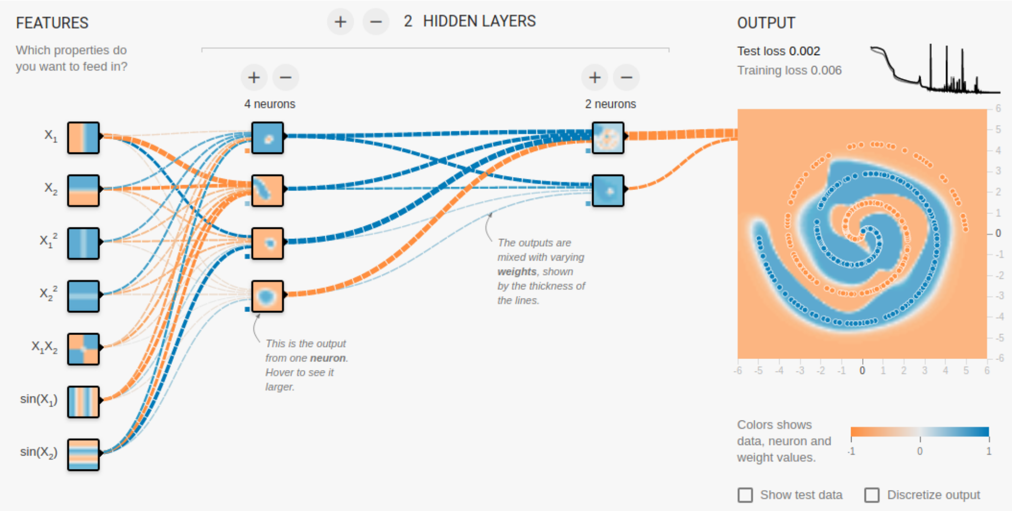
TensorFlow Playground — Spiral



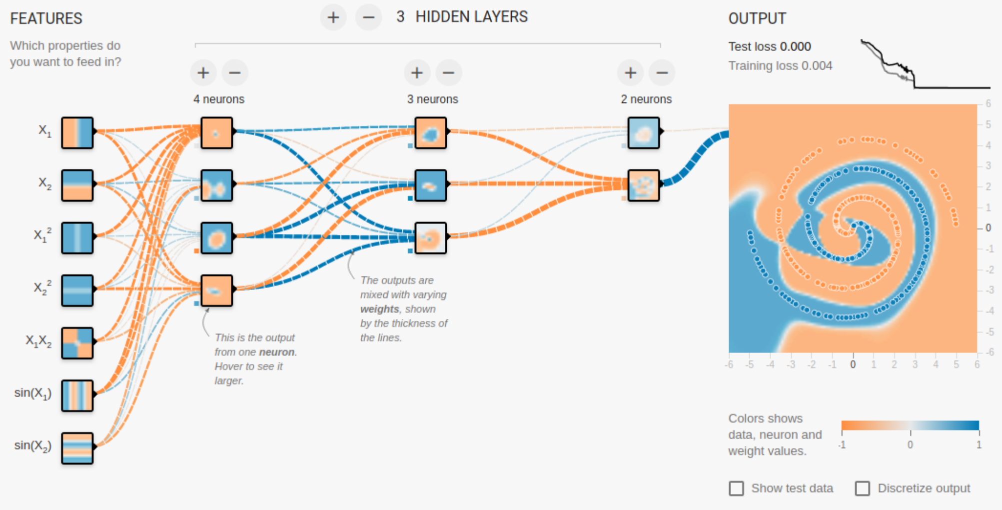
TensorFlow Playground — Spiral



TensorFlow Playground — Spiral



TensorFlow Playground — Spiral



Neural Nets and Abstraction

Deep neural networks learn hierarchical feature representations

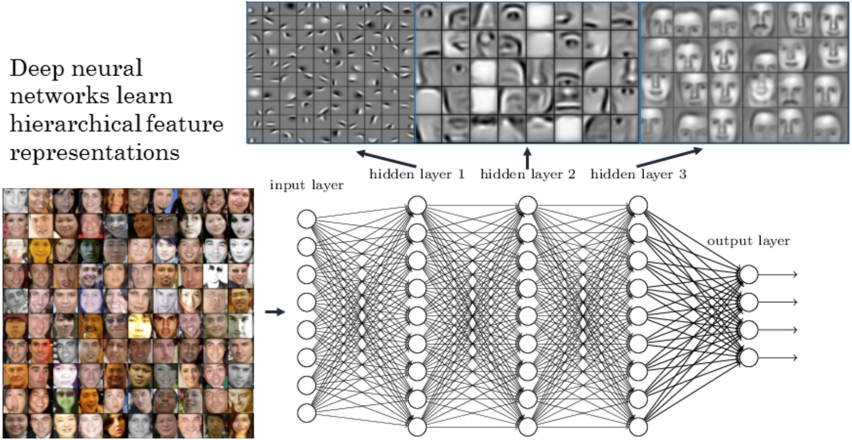


Figure: https://www.rsipvision.com/exploring-deep-learning/

Probabilistic Prediction

So far we've just looked at predictors that give us $\mathbb{E}\left[y^* | h(\mathbf{x}^*, \hat{\theta})\right]$ — don't explicitly model uncertainty.

Maximum Likelihood methods:

$$p(y^* | h(\mathbf{x}^*, \hat{\theta}))$$

model the uncertainty in the **targets** (point estimate parameters).

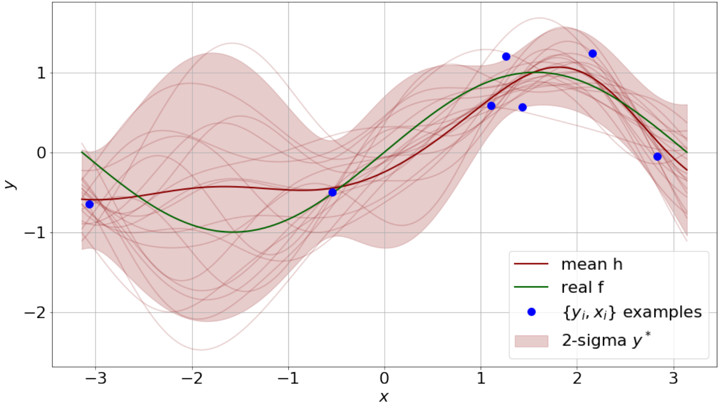
Bayesian methods:

$$\frac{1}{S} \sum_s p(y^* | h(\mathbf{x}^*, \theta_s)), \qquad \theta_s \sim p(\theta | \{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots\})$$

also model the uncertainty in the **model parameters**.

Probabilistic Prediction

Here is the same sin function fit using a Bayesian predictor (GP):



Note: where there isn't much data, uncertainty grows …

Why Probabilistic Prediction?

Great for decision making, e.g.

- *Don't use this prediction because the model is uncertain* — good when predicting something about people.
- *Getting more data* — where should I sample next to make the model more confident?

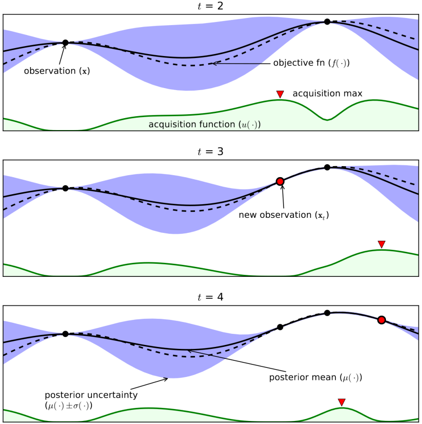
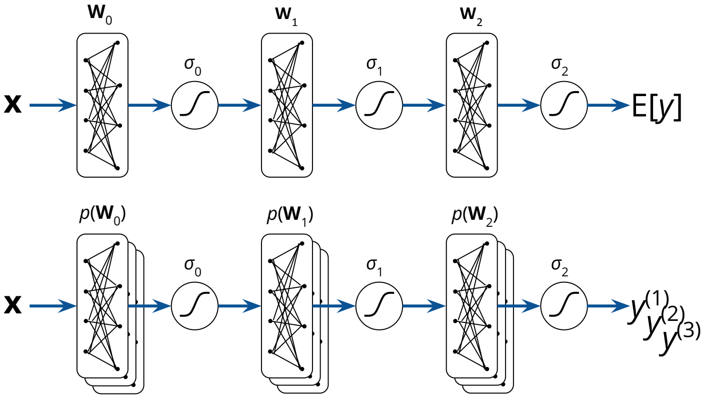


Figure: Bayesian optimisation

Bayesian Neural Nets



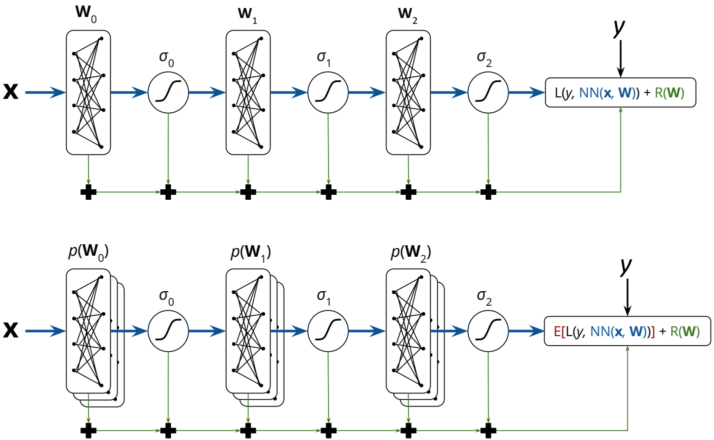
Bayesian NNs propagate **random samples of their weights** through the network, making a stochastic neural network.

Bayesian Neural Nets

Learning involves **function composition** (evaluating the NN), and **accumulation** (weight regularisation).

For a BNN, its samples have to be **built into the computational graph** since they are part of the **learning objective**.

This motivates the need for a specialised framework for BNNs.



Aboleth

Why do we need another NN framework?

- **TensorFlow, PyTorch, MXNet, …** — too low level (a lot of boiler-plate)
- **Keras** — not inherently Bayesian
- **Edward, PyMC3** — probabilistic programming, a bit too general
- **ZhuSuan** — Like Edward, but with less functionality and worse design

So, there isn't really a *simple to use and extendable* library specifically for Bayesian NNs (…in python 3)

Aboleth Features

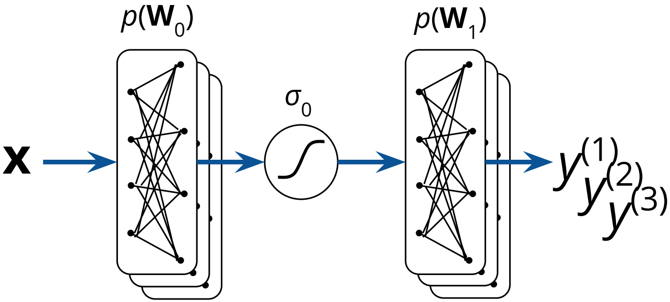
- Built on TensorFlow (great for deployments, monitoring etc)
- Bayesian layers (Dense, convolutional, embedding)
- Simple interface, simple to extend/interoperate with underlying TensorFlow
- Large scale Gaussian process approximation
- Multiple inputs: imputing layers, embedding layers etc
- Compatible with Keras
- **Stochastic variational Bayes**[1] inference (and SGD)

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

Interface comparison

Let's implement a one hidden layer Bayesian NN:

- $\mathbf{x}$ and y are one dimensional
- $\mathbf{W}_0 \in \mathbb{R}^{1 \times 20}$
- $\mathbf{W}_1 \in \mathbb{R}^{20 \times 1}$



Interface comparison — Edward

Let's implement a one hidden layer Bayesian NN in Edward (`http://edwardlib.org/`):

```python
from edward.models import Normal

W_0 = Normal(loc=tf.zeros([1, 20]), scale=tf.ones([1, 20]))
W_1 = Normal(loc=tf.zeros([20, 1]), scale=tf.ones([20, 1]))

def neural_network(x):
    h = tf.tanh(tf.matmul(x, W_0))
    h = tf.matmul(h, W_1)
    return tf.reshape(h, [-1])

y = Normal(loc=neural_network(x_train), scale=0.1)
```

Interface comparison — Edward

Continuing …

```
import edward as ed

qW_0 = Normal(loc=tf.get_variable(”qW_0/loc”, [1, 20]),
          scale=tf.nn.softplus(tf.get_variable(”qW_0/scale”, [1, 20])))
qW_1 = Normal(loc=tf.get_variable(”qW_1/loc”, [20, 1]),
          scale=tf.nn.softplus(tf.get_variable(”qW_1/scale”, [20, 1])))

inference = ed.KLqp({W_0: qW_0, W_1: qW_1}, data={y: y_train})
inference.run(n_iter=1000)
```

Interface comparison — Edward

Some remarks:

- Quite a general probabilistic framework (great for prototyping Bayesian models)
- As a result, probably requires a bit too much boilerplate for BNNs
- Make sure you get the dimensions of your layers right!
- PyMC3 is quite similar

Interface comparison — ZhuSuan

Let's implement a one hidden layer Bayesian NN in ZhuSuan (`https://github.com/thu-ml/zhusuan`)

Too long — click here...

Some remarks:

- What is the probability of getting this right the first time?
- Why wouldn't I use Edward or PyMC3?

Interface comparison — Aboleth

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

# Construct the network, no data needed
net = (
  ab.InputLayer(name="X", n_samples=5) »   # how we assign data, and # samples
  ab.DenseVariational(output_dim=20) »
  ab.Activation(tf.tanh) »
  ab.DenseVariational(output_dim=1)
  )

# Build the computational graph for the net, attach data x_train
nn, reg = net(X=x_train)

# Now make the training objective, attach targets
likelihood = tf.distributions.Normal(loc=nn, scale=0.1).log_prob(y_train)
loss = ab.elbo(likelihood, reg, N_training)

# Standard TensorFlow training code here
```

Interface comparison — Aboleth

- » implements the NN **function composition**, and **regularisation accumulation**
  - Tony has told us this is an instance of a **writer monad**
  - Each layer composes itself with previous layers, and accumulates its complexity penalty
- We don't need to know the shape of the input (unlike Keras), or the inputs to any layers, only the output shapes!
  - the input shapes are lazily evaluated when data/placeholders are input

Layers in Aboleth

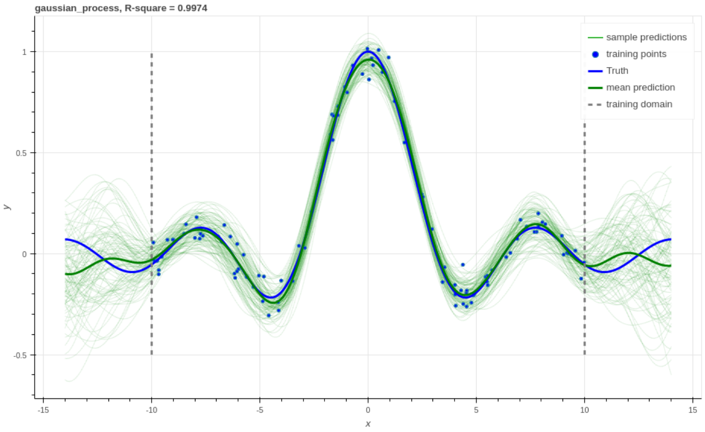
```python
class Layer:
    """Layer base class."""

    def __call__(self, X):
        """Construct the subgraph for this layer."""
        Net, KL = self._build(X)
        return Net, KL

    def _build(self, X):
        """Implement graph construction. Should be over-ridden."""
        return X, 0.0

    def __rshift__(self, other):
        """Implement layer composition, other(self(x))."""
        return LayerComposite(self, other)
```

- Usually you just need to subclass `Layer`, and implement `_build`, `__init__`.
- We also have `MultiLayers`, which take in key-word argument pairs

Aboleth Examples

Lets look at how we can use Aboleth for regression, we have some docs here:

`http://aboleth.readthedocs.io/en/stable/tutorials/some_regressors.html`



Use Aboleth for your *whole* pipeline

Learn values to impute missing data as part of the network

```
# Construct the network, no data needed
data_input = ab.InputLayer(name=”X”, n_samples=5)
mask_input = ab.MaskInputLayer(name=”M”)

net = (
    ab.LearnedScalarImpute(data_input, mask_input) »
    ab.DenseVariational(output_dim=1, full=True)
)

# Build the computational graph for the net, attach data X, M
nn, reg = net(X=x_data, M=x_missing_mask)
# ...
```

- We now have two input layers, one for data, and one for a missingness mask
- We then attach the data and mask

Combine Networks

```
# Ordinal data
ord_net = (
    ab.InputLayer(name="Xord", n_samples=5) »
    ab.DenseVariational(output_dim=10)
)

# Categorical data
cat_net = (
    ab.InputLayer(name="Xcat", n_samples=5) »
    ab.EmbedVariational(output_dim=10, n_categories=100)
)

# Join them
net = (
    ab.Concat(cat_net, ord_net) »
    ab.Activation(tf.nn.relu)  »
    ab.DenseVariational(output_dim=1)
)

nn, reg = net(Xcat=x_categorical, Xord=x_ordinal)
```

- We can create separate embedding pipelines then join (concatenate) them
- We can add imputing to this
- We also have tools to replicate the categorical embedding for more categorical features

Please dive in!

- Repo: `https://github.com/data61/aboleth`
- Docs: `http://aboleth.readthedocs.io/en/stable/?badge=stable`
- We've been using it for > 6 months on most projects
- Easy to deploy on Bracewell
- Lots of issues that we need help with!

Thanks!

Contributors: Dan Steinberg, Lachlan McCalman, Louis Tiao, Simon O'Callaghan, Alistair Reid
{daniel.steinberg, lachlan.mccalman}@data61.csiro.au