Edward: Getting Started

Michael L. Thompson

May 1, 2017

Contents

Preface 1
Setup 2
Build Data 3
Define Bayesian Neural Network 3
Augment Computational Graph with Approximation to Posterior Distribution 4
Visualize the Prior Fit 5
Train the Model 6
Visualize the Posterior Fit 6
Conclusion 7

Preface

This is an R implementation of the Edward tutorial "Getting Started" (jupyter notebook).

Done per Edward's license instructions, i.e., Apache License, version 2.0.

Assuming R and RStudio already installed, this implementation (under Windows 10) was made possible by installing Python and the TensorFlow and Edward modules, and then the `reticulate` and `tensorflow` packages in R, per the instructions at the following links:

- Python setup:
  - Python: Python, Download the latest version for Windows
  - TensorFlow: Installing TensorFlow on Windows
  - Edward: Edward, Getting Started
  - Also `pip`-ed `numpy` and `jupyter`.
- R setup:
  - Package `reticulate`: R Interface to Python
  - Package `tensorflow`: TensorFlow for R

I'm absolutely thrilled all of this worked on the first shot!

(*This notebook also uses R packages* ***`magrittr`*** *&* ***`tidyverse`****, which are my go-to tools for data science in R.*)

But, why bother?...

Although the performance is nothing to brag about, this does give someone like me – who is an old hand in R programming – a nice way to test advanced probabilistic programming and deep learning ideas. I can do it without continually referencing the documentation for Python & its various modules each time I need to manipulate the data structures and to implement the algorithms that I can do almost unconsciously in R. So my productivity in ideation is greatly increased by using this Edward-&-Tensorflow-via-R implementation. I don't need to think nearly as much about the "how to do" and can concentrate on the "what to do" of my ideas.

All that said, I'm still compelled to ultimately do a lot more in Python directly. Guess, this is just my security-blanket/training-wheels phase ....

Michael L. Thompson

Setup

Load the R packages and the Python modules we need.

Note that in all that follows, the R code explicitly specifies integer constants as being long integers by appending "L" to the number. This is necessary for proper interfacing with Python using the `reticulate` package in R.

```
library( magrittr )
library( tidyverse )
library( reticulate )
library( tensorflow )

ed   <- import( module = "edward" )
np   <- import( module = "numpy" )

# Assign the Edward models used in the script.
Normal <- ed$models$Normal
```

Function `build_toy_dataset( N, amplitude, period, noise_std )`

This function implements the "true" model directly in R, returning a `data_frame` object:

$$
\begin{aligned}
x &\in [-3, 3] \\
y(x) &= A cos(\frac{2\pi}{\omega} x) + \epsilon \\
\epsilon &\sim N(0, \sigma_\epsilon)
\end{aligned}
$$

where A is the amplitude, ω is the period, ϵ is zero-mean normally distributed noise, and σ_ϵ is the standard deviation of the noise.

```
build_toy_dataset <- function( N = 50L, amplitude = 1.0, period = 2.0*pi/3.0, noise_std = 0.1 ){
  x = seq(-3, 3, length.out = N)
  y = cos( 3*x ) + rnorm(n = N, mean = 0, sd = noise_std )
  return( data_frame( x = x, y = y ) )
}
```

Function `neural_network( x, W_0, W_1, b_0, b_1 )`

This function implements the 2-layer neural network as nodes in a TensorFlow computational graph:

$$
\begin{aligned}
\mathbf{x} &\in \Re^{(D_x \times 1)} \\
\mathbf{W}_0 &\in \Re^{(D_x \times N_h)}; \mathbf{b}_0 \in \Re^{(N_h \times 1)} \\
\mathbf{W}_1 &\in \Re^{(N_h \times D_y)}; \mathbf{b}_1 \in \Re^{(D_y \times 1)} \\
\mathbf{f} &\in \Re^{(D_y \times 1)} \\
\mathbf{h}(\mathbf{x}) &= \tanh(\mathbf{x}^\intercal \mathbf{W}_0)^\intercal + \mathbf{b}_0 \\
\mathbf{f}(\mathbf{x}) &= (\mathbf{h}^\intercal \mathbf{W}_1)^\intercal + \mathbf{b}_1
\end{aligned}
$$

where D_x is the dimensionality of the x vector (i.e., the number of input features), D_y is the dimensionality of the y vector (i.e., the number of output responses), and N_h is the number of hidden nodes.

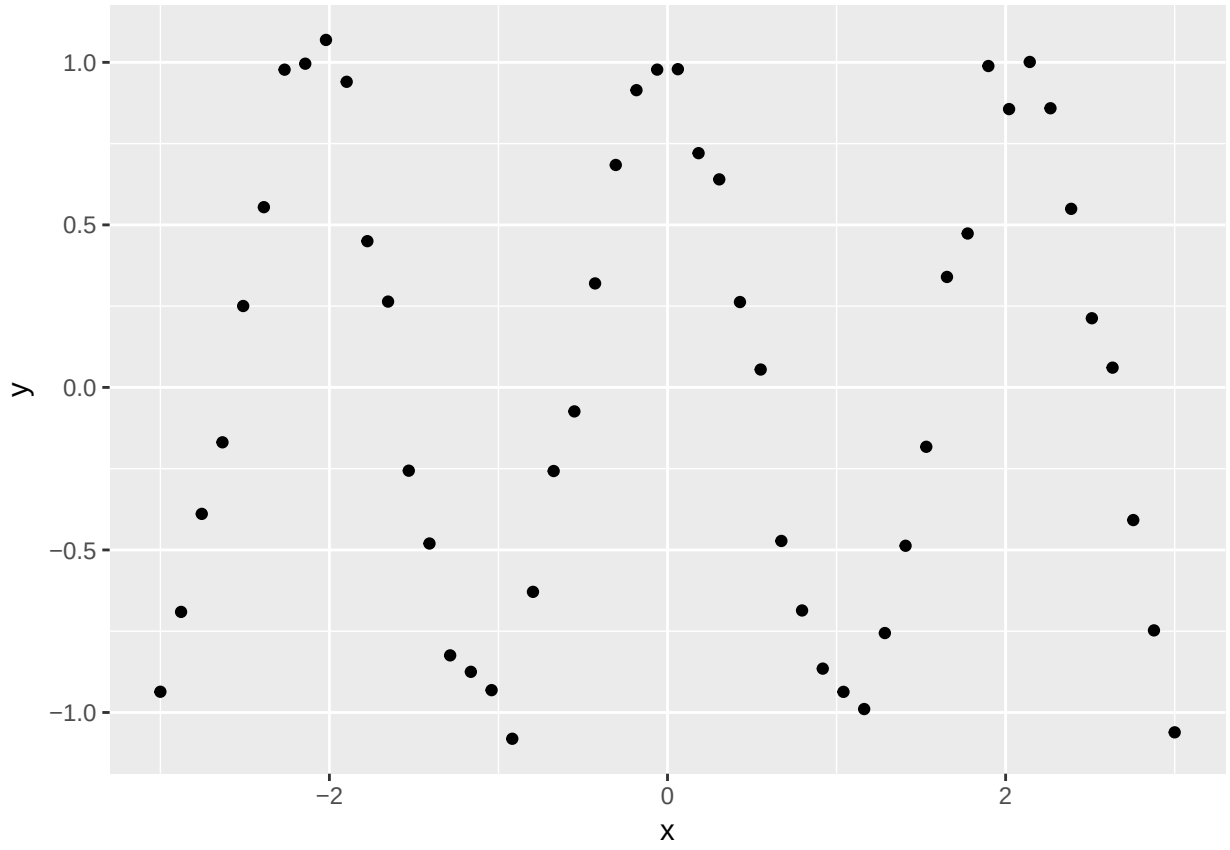
In our example, $D_x = 1$, $D_y = 1$, and $N_h = 9$.

```
neural_network <- function( x, W_0, W_1, b_0, b_1 ){
  h <- tf$tanh( tf$matmul(x, W_0) + b_0 )
  h <- tf$matmul( h, W_1 ) + b_1
  return( tf$reshape( h, array(-1L,1L) ) )
}
```

Build Data

First, simulate a toy dataset of 50 observations with a cosine relationship.

```
# Prep the TensorFlow default computational graph and seed the random number generator for Edward.
tf$reset_default_graph()
ed$set_seed( 42L )

N <- 50L # number of data points
D <-  1L # number of features

xy_train <- build_toy_dataset( N )
quickplot(x,y,data=xy_train)
```



Define Bayesian Neural Network

Next, define a two-layer Bayesian neural network. Here, we define the neural network manually with `tanh` nonlinearities.

The weights receive Gaussian priors, and the neural network will serve as the model $\mathbf{f}(\mathbf{x})$ in the Gaussian likelihood function:

$$
\begin{aligned}
W_{0,i,j} &\sim N(0,1); \forall i \in \{1,...,D_x\} \ , \forall j \in \{1,..,N_h\} \\
b_{0,j} &\sim N(0,3) \forall j \in \{1,..,N_h\} \\
W_{1,i,j} &\sim N(0,1); \forall i \in \{1,...,N_h\} \ , \forall j \in \{1,..,D_y\} \\
b_{1,j} &\sim N(0,1) \forall j \in \{1,..,D_y\} \\
\hat{\mathbf{y}}(\mathbf{x}) &= \mathbf{f}(\mathbf{x}) \\
\mathbf{y} &\sim N(\hat{\mathbf{y}}(\mathbf{x}), \mathbf{\Sigma}_\epsilon)
\end{aligned}
$$

In our example, since $D_y = 1$, we have a univariate distribution $y \sim N(\hat{y}(x), \sigma_\epsilon^2)$, where scalar $\sigma_\epsilon^2 = \mathbf{\Sigma}_\epsilon$.

```
n_hidden <- 9L
# We use Edward model Normal for the priors of all of the unknowns.
W_0 <- Normal( loc = tf$zeros(     c(D, n_hidden) ), scale = tf$ones(     c(D, n_hidden) ) )
b_0 <- Normal( loc = tf$zeros( array(n_hidden,1L) ), scale = 3*tf$ones( array(n_hidden,1L) ) )

W_1 <- Normal( loc = tf$zeros( c(n_hidden, 1L) ), scale = tf$ones( c(n_hidden, 1L) ) )
b_1 <- Normal( loc = tf$zeros(    array(1L,1L) ), scale = tf$ones(    array(1L,1L) ) )

x  <- tf$cast(matrix(xy_train$x,ncol=1L), tf$float32)
y  <- Normal(loc=neural_network(x, W_0, W_1, b_0, b_1),scale=0.1 * tf$ones(N))
```

Augment Computational Graph with Approximation to Posterior Distribution

Next, make inferences about the model from data. We will use variational inference. Specify a normal approximation over the weights and biases.

The approximations to the posterior distributions of all of the weights will be Gaussians:

```
# $$
# \begin{align*}
#    q_{W0,i,j} &\sim N(\mu_{W0;i,j},\text{g}(\sigma'_{W0;i,j}))\text{;  } \forall i\in \{1,...,D_x\} \
#    q_{b0,j} &\sim N(\mu_{b0;j},\text{g}(\sigma'_{b0;j}))\text{;  } \forall j\in \{1,..,N_h\}\\
#    q_{W1,i,j} &\sim N(\mu_{W1;i,j},\text{g}(\sigma'_{W1;i,j}))\text{;  }\forall i\in \{1,...,N_h\} \t
#    q_{b1,j}&\sim N(\mu_{b1;j},\text{g}(\sigma'_{b1;j}))\text{;  }  \forall j\in \{1,..,D_y\}\\
#    \text{where} \\
#    \text{g}(u) &\equiv \log(1 + \exp(u) )\\
# \end{align*}
# $$
```

(Above, g(u) is the "softplus" function described at the link below.)

Inference will find optimal values for the $\mu_{...}$ and $\sigma'_{...}$ parameters.

```
qW_0 <- Normal(
  loc   = tf$Variable(tf$random_normal(c(D, n_hidden))),
  scale = tf$nn$softplus(tf$Variable(tf$random_normal(c(D, n_hidden))))
)
qb_0 = Normal(
  loc   = tf$Variable(tf$random_normal(array(n_hidden,1L), stddev = 3*tf$ones( array(n_hidden,1L) ) )),
  scale = tf$nn$softplus(tf$Variable(tf$random_normal(array(n_hidden,1L))))
)
```

```
qW_1 = Normal(
  loc   = tf$Variable(tf$random_normal(c(n_hidden, 1L))),
  scale = tf$nn$softplus(tf$Variable(tf$random_normal(c(n_hidden, 1L))))
)
qb_1 = Normal(
  loc   = tf$Variable(tf$random_normal(array(1L,1L))),
  scale = tf$nn$softplus(tf$Variable(tf$random_normal(array(1L,1L))))
)
```

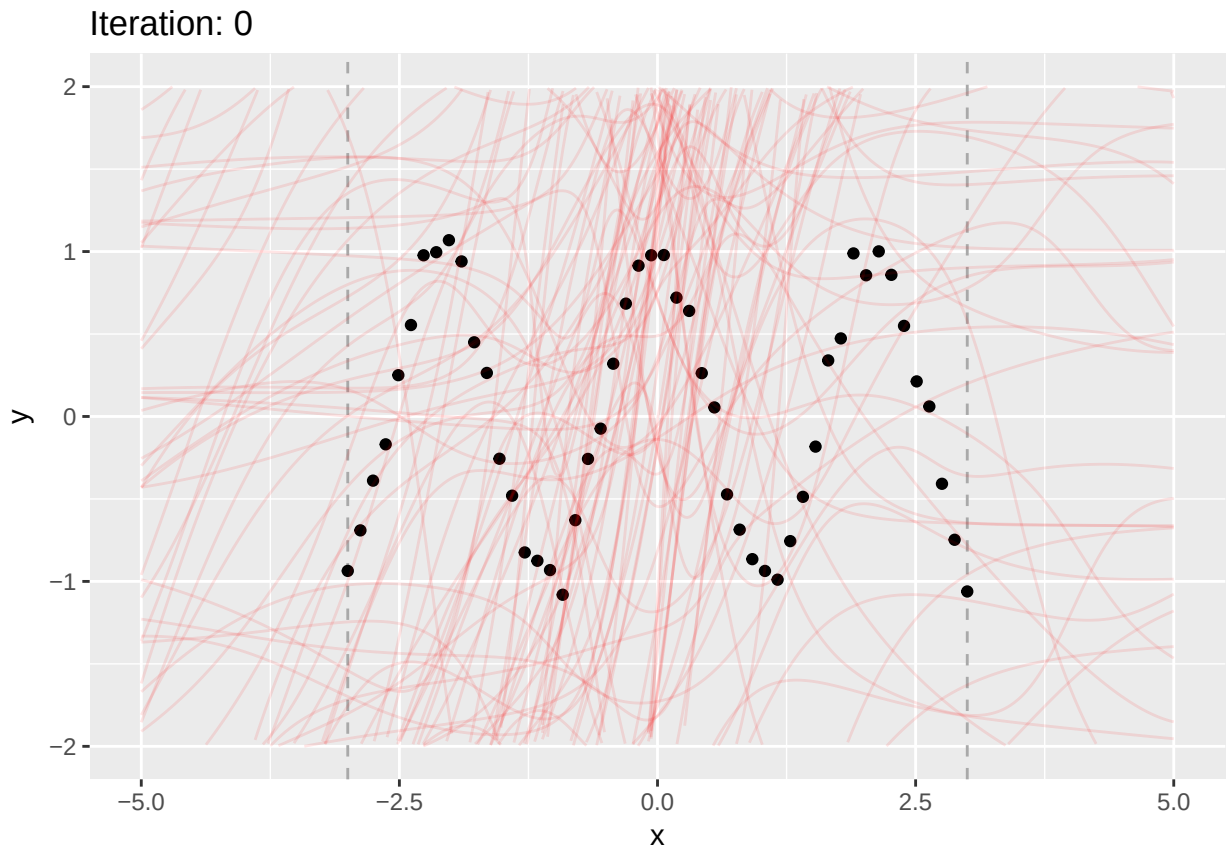
Defining `tf$Variable` allows the variational factors' parameters to vary. They are initialized randomly. The standard deviation parameters are constrained to be greater than zero according to a softplus transformation.

Visualize the Prior Fit

```
# Sample functions from variational model to visualize fits.

inputs <- seq(-5, 5, length.out=400)
x <- tf$expand_dims(inputs, 1L)

N_sample <- 100L
mus <- tf$stack(
  lapply(
    seq_len( N_sample ),
    function(i) neural_network(x, qW_0$sample(), qW_1$sample(), qb_0$sample(), qb_1$sample())
  )
)
```

Execute the computational graph in the default Edward session to compute random realizations from the prior distribution.

```
# FIRST VISUALIZATION (prior)

sess <- ed$get_session()
#sess$run(tf$global_variables_initializer())
tf$global_variables_initializer()$run( )
outputs <- mus$eval()
prior <- data_frame( x = inputs ) %>%
  bind_cols( as_data_frame( as.matrix(t(outputs)) ) ) %>%
  gather(key=iteration,value=y,-x)

plt_prior <- xy_train %>%
  {
    ggplot(., aes( x=x,y=y ) ) +
      geom_vline( xintercept = range((.)$x), linetype = 2, color = 'darkgray' ) +
      geom_point() +
      geom_line( data=prior,aes(x=x,y=y,group=iteration), na.rm=TRUE, alpha=0.1, color='red') +
      ggtitle( "Iteration: 0" ) +
      lims( x = c(-5,5),y = c(-2,2) ) +
      theme( legend.position = "none")
  }
print( plt_prior )
```



Train the Model

Now, run variational inference with the Kullback-Leibler divergence in order to infer the model's latent variables given data. We specify 3000 iterations.

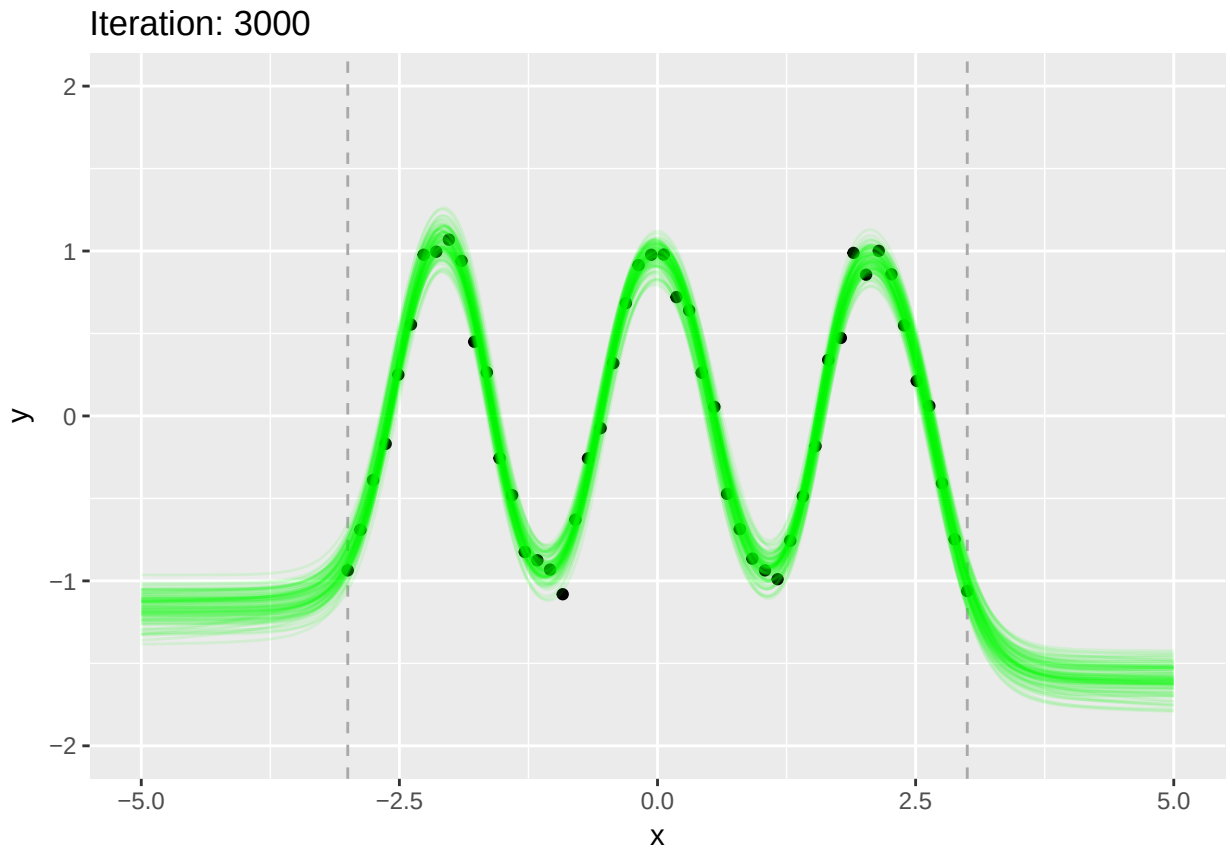
```
n_iter <- 3000L
inference <- ed$KLqp(
  dict( W_0 = qW_0, b_0 = qb_0, W_1 = qW_1, b_1 = qb_1),
  data = dict( y = tf$cast(xy_train$y, tf$float32)$eval() )
)

# KEYS:
# Arguments needed to be integer (with "L" designation) to get this to work.
# Also, it wouldn't get past the first printout, so had to set
# `n_print` to zero to kill all printing.
# Should provide feedback to the Edward team:
# Provide a way to get output without the special
# character printed progress bar.
inference$run( n_iter = n_iter, n_samples = 5L, n_print = 0L )
```

Visualize the Posterior Fit

Finally, criticize the model fit. Bayesian neural networks define a distribution over neural networks, so we can perform a graphical check. Draw neural networks from the inferred model and visualize how well it fits

the data.

```
# SECOND VISUALIZATION (posterior)
outputs <- mus$eval()
post <- data_frame( x = inputs ) %>%
  bind_cols( as_data_frame( t(outputs) ) ) %>%
  gather(key=iteration,value=y,-x)

plt_post <- xy_train %>%
  {
    ggplot(., aes( x=x,y=y ) ) +
      geom_vline( xintercept = range((.)$x), linetype = 2, color = 'darkgray' ) +
      geom_point() +
      geom_line( data=post,aes(x=x,y=y,group=iteration), na.rm=TRUE, alpha=0.1, color='green') +
      ggtitle( sprintf("Iteration: %d", n_iter) ) +
      lims( x = c(-5,5),y = c(-2,2) ) +
      theme( legend.position = "none")
  }
print( plt_post )
```




```
sess$close()
```

Conclusion

The model has captured the cosine relationship between `x` and `y` in the observed domain.

To learn more about **Edward**, delve in!

If you prefer to learn via examples, then check out some tutorials.