The Wheat Yield Dataset

Presenters: Jay Barber and Dan Pagendam

The Wheat Yield Dataset

- Exercise developed by Chris Wikle and Dan Pagendam (2019).
- Dataset created by Dan Pagendam and Josh Bowden at CSIRO.
- The data consists of simulated wheat yields from a farm in Dalby, Queensland using the model APSIMX.
- The predictors are:
  - summary statistics for the amount of rainfall in each of 52 weeks in the year.
  - degree days in each of the 52 weeks in the year.
  - cumulative evaporation in each of the 52 weeks of the year.
  - the “thermal time” of the wheat at each of the 52 weeks in the year.

The Wheat Yield Dataset

(continued)

- The predictors are:
  - the amount of Nitrogen fertiliser applied at planting.
  - the amount of Nitrogen fertiliser applied as top-up.
  - the day of year that the crop was planted.
  - the planting density of seed.

The Wheat Yield Dataset

(continued)

- The response variables are:
  - wheat grain yield.
  - grain size.
  - grain protein content.
  - wheat total weight.

Loading the Wheat Dataset

```
library(remotes)
remotes::install_github("dpagendam/deepLearningRshort")
library(deepLearningRshort)
#For those using Colab run: install.packages("keras")
data("wheat")
```

- The dataset consists of four objects:
  - trainData_X
  - trainData_Y
  - testData_X and
  - testData_Y.
- The dataset was created from 10,000 simulations of wheat growth under randomly generated meteorological conditions and management (planting and fertiliser application).

Loading the Wheat Dataset

- The training and test sets were determined by randomly allocating approximately 10% of the simulations to the test/validation set and the remaining 90% to the training set.

```
dim(trainData_X)
```

```
## [1] 8993  212
```

```
dim(testData_X)
```

```
## [1] 1007  212
```

Scaling the Data for Training

- For gradient descent to work well, we need to scale the input data to the interval [0,1].

```
rescaleCols <- function(rowX, colMins, colMaxs)
{
  r <- (rowX - colMins)/(colMaxs - colMins)
  r[is.nan(r)] <- 0
  return(r)
}
```

- Obtain the column mins and maxs of trainData_X

```
colMinsX <- apply(trainData_X, 2, min)
colMaxsX <- apply(trainData_X, 2, max)
```

Scaling the Data for Training

- Now scale the input data to the interval [0,1]

```
trainData_X_scaled <- t(apply(trainData_X, 1, rescaleCols,
                              colMinsX, colMaxsX))
testData_X_scaled <- t(apply(testData_X, 1, rescaleCols,
                             colMinsX, colMaxsX))
```

Wrangling the output data

- For this exercise we will focus on modelling the total plant biomass.

```
trainData_Y = matrix(trainData_Y[, "wheatTotalWeight"], ncol = 1)
testData_Y = matrix(testData_Y[, "wheatTotalWeight"], ncol = 1)
```

Building a FFNN model

- Keras lets us define a model “sequentially” starting at the inputs, and working downwards to the outputs.
- Let’s use 3 hidden layers, with 64 nodes per layer and stick with the rectified linear unit (ReLU) activation function.

```
library(keras)
model <- keras_model_sequential()
model %>% layer_dense(units = 64, activation = "relu",
          input_shape = ncol(trainData_X_scaled)) %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 1)
```

Compiling the model

- We'll use the RMSProp optimiser with a relatively small learning rate.
- To start with, we will use the built in Mean Squared Error (MSE) loss function.
- Keras needs us to "compile" the model with the optimisation algorithm and loss function.

```
model %>% compile(
  loss = "mse",
  optimizer = optimizer_rmsprop(learning_rate = 0.0001)
)
```

Fitting the model

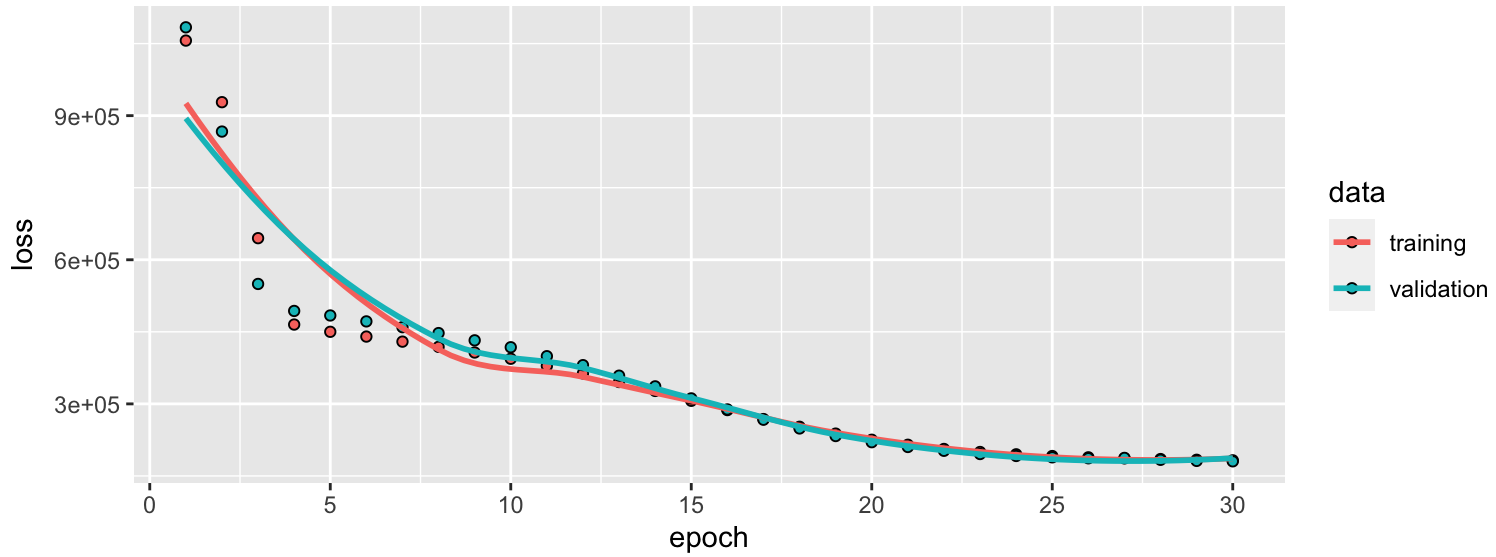
- We are now ready to fit the model.
- We define the training data, but also out-of-sample validation data that we can use to check how well the model generalises.
- When training the neural network, we use batches of data (32 samples here) to estimate the gradient of the loss function w.r.t the parameters.
- We also need to specify the number of epochs.
- Because we chose a small learning rate, we'll probably need to use more epochs to find the optimal fit.

```
history <- model %>% fit(
  x = trainData_X_scaled, y = trainData_Y,
  epochs = 30, batch_size = 32,
  validation_data = list(testData_X_scaled, testData_Y)
)
```

Fitting the model

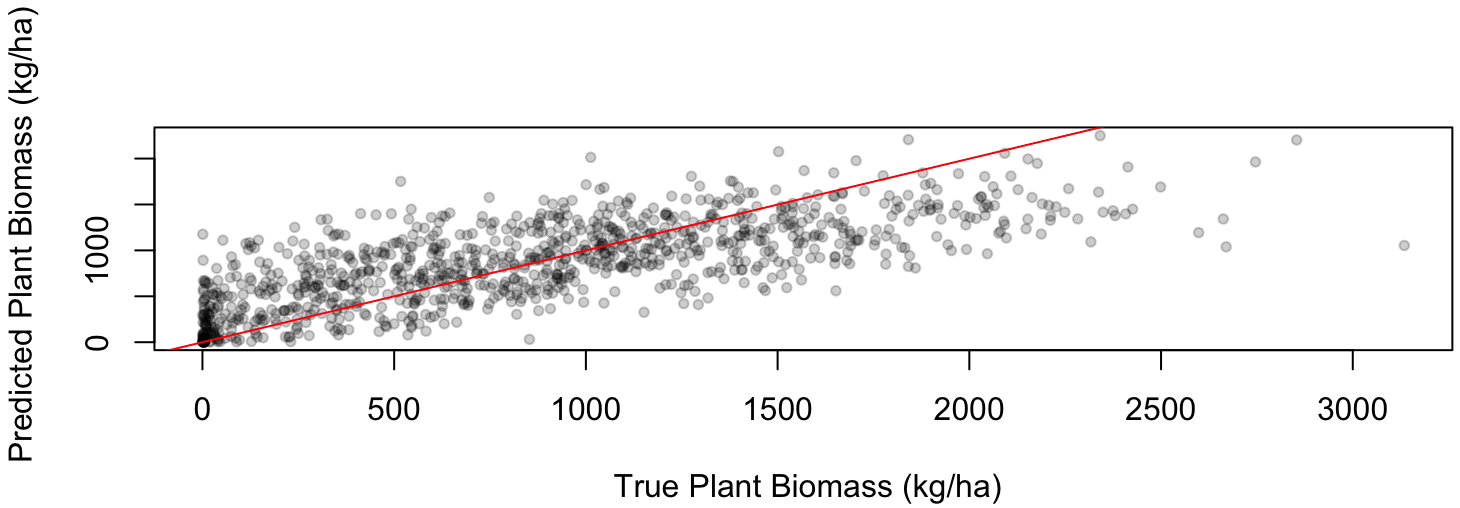
- We want to be careful not to overfit the model.
- We can often see signs of overfitting in the training history.

```
plot(history)
```



Fitting the model

```
pred_test <- predict(model, testData_X_scaled)
plot(testData_Y, pred_test, xlab = " True Plant Biomass (kg/ha)",
     ylab = " Predicted Plant Biomass (kg/ha)", pch = 20,
     col = adjustcolor("black", 0.2))
abline(0, 1, col = "red")
```



Custom loss functions

- We just trained a model using the build in “mse” loss function. This minimises mean squared error and so is analogous to linear regression where we assumed the error to be homoskedastic.
- One of the most powerful things you can do is to make custom loss functions for your model.
- This provides a way for you to use Deep Neural Networks in more statistical ways than is typical in Machine Learning applications.
- We’re going to create a loss function that is equal to the negative of the log-likelihood under the assumption that our predicted wheat yields can be modelled as having a log-normal distribution.

Custom loss functions

- all custom loss functions have the same two inputs:
  - y_true: the true values of your output which you provide as data when training.
  - y_pred: the predictions that come out of your neural network model.

```
myLossFunction <- function(y_true, y_pred)
{
  #code goes here
}
```

Custom loss functions

- Important notes:
  - when writing your code, you need to remember that y_true and y_pred are tensors.
  - y_true and y_pred don't have to have the same dimension.
  - the first dimension of both the input tensors is equal to the batch size.
  - there is very little documentation on how to write custom loss functions.
  - try to use the Keras backend functions as much as possible and avoid using R functions (they may not work).
  - if a Keras function has the "axis" argument, it is asking you which dimension you want to "apply"" the function to.

Custom loss functions

```
negLL_logNormal <- function(y_true, y_pred)
{
  K <- backend()
  # Set up muMask and sigma Mask as 2 x 1 matrices.
  muMask <- K$constant(matrix(c(1, 0), 2, 1), shape = c(2, 1))
  sigmaMask <- K$constant(matrix(c(0, 1), 2, 1), shape = c(2, 1))

  # Extract the first and second columns
  mu <- K$dot(y_pred, muMask)
  sigma <- K$exp(K$dot(y_pred, sigmaMask))

  # Use mu and sigma as parameters describing log-normal distributions
  logLike <- -1*(K$log(y_true) + K$log(sigma)) -
  0.5*K$log(2*pi) -
  K$square(K$log(y_true) - mu)/(2*K$square(sigma))
  -1*(K$sum(logLike, axis = 1L))
}
```

Building a FFNN model

- Based on the custom loss function we just created, our model requires two outputs:
  - one for mu (location parameter of the log-normal).
  - one for sigma (scale parameter of the log-normal).
- Let's use 3 hidden layers, with 64 nodes per layer and stick with the rectified linear unit (ReLU) activation function.
- Notice, that we now have two nodes / units in the output layer of the network.

```
model <- keras_model_sequential()
model %>% layer_dense(units = 64, activation = "relu",
          input_shape = ncol(trainData_X_scaled)) %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 64, activation = "relu") %>%
layer_dense(units = 2)
```

Compiling the model

- We'll use the RMSProp optimiser with a relatively small learning rate again.
- This time, we specify our custom loss function when compiling the model.

```
model %>% compile(
  loss = negLL_logNormal,
  optimizer = optimizer_rmsprop(learning_rate = 0.0001)
)
```

Fitting the model

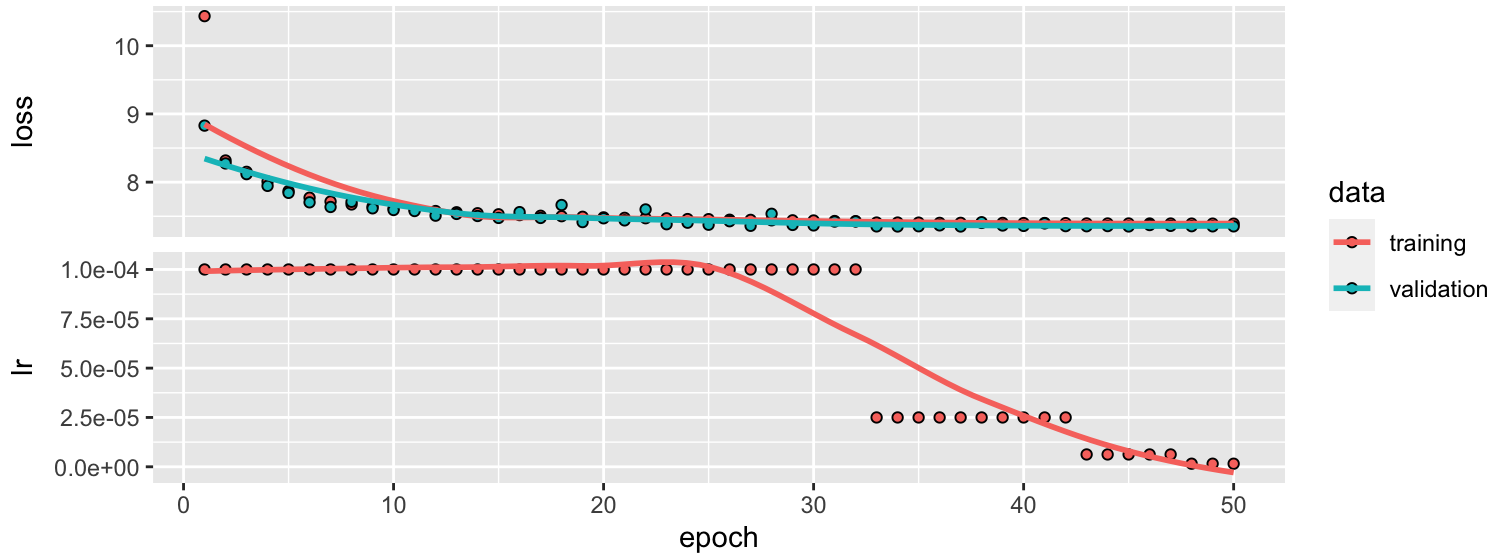
- We're also going to introduce a callback into our training procedure.
- This will reduce the learning rate when we stop seeing a reduction in the validation loss.
- A smaller learning rate means smaller changes to the parameters, so we can think of this as fine-tuning our parameters with more and more epochs.

```
history <- model %>% fit(
  x = trainData_X_scaled, y = trainData_Y,
  epochs = 50, batch_size = 32,
  validation_data = list(testData_X_scaled, testData_Y),
  callbacks = list(callback_reduce_lr_on_plateau(monitor = "val_loss",
              factor = 0.25, patience = 5))
)
```

Fitting the model

- We want to be careful not to overfit the model.
- We can often see signs of overfitting in the training history.

```
plot(history)
```



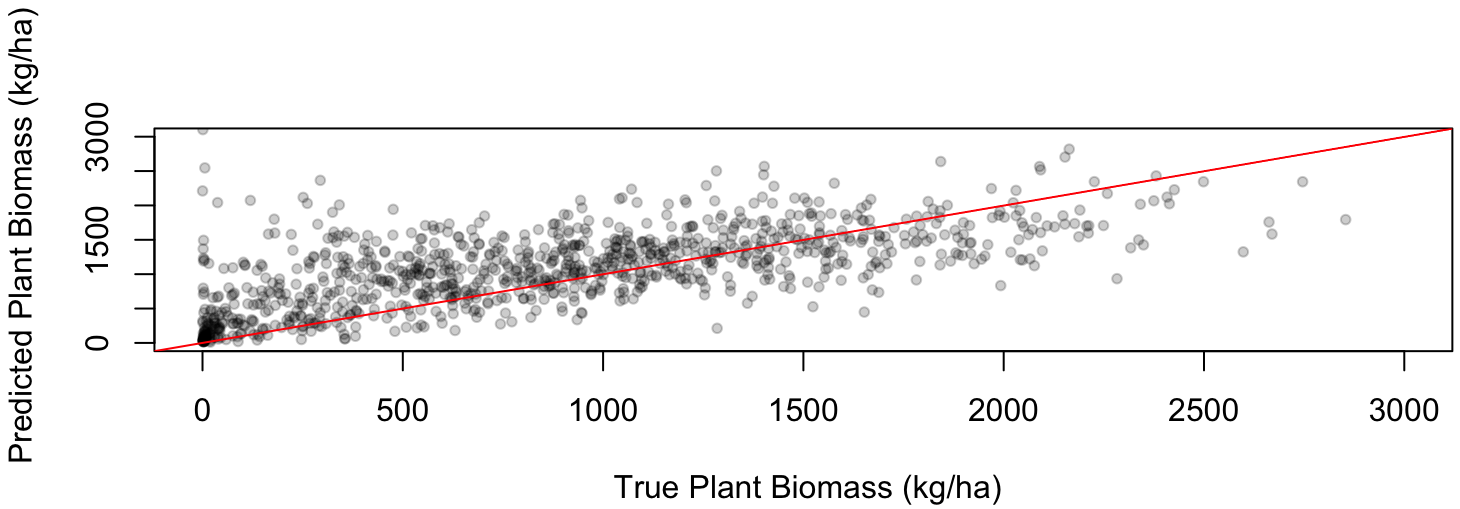
Assessing Model Performance

- The model outputs two quantities for each prediction.
- These are the mu and log(sigma) parameters for a log-normal predictive density.
- We can compare the mean of the log-normal to the true yield.

```
yhat <- predict(model, testData_X_scaled)
mu <- yhat[, 1]
sigma <- exp(yhat[, 2])
pred_mean <- exp(mu + 0.5*sigma^2)
```

Assessing Model Performance

```
plot(testData_Y, pred_mean, xlab = " True Plant Biomass (kg/ha)",
     ylab = " Predicted Plant Biomass (kg/ha)", xlim = c(0, 3000),
     ylim = c(0, 3000), pch = 20, col = adjustcolor("black", 0.2))
abline(0, 1, col = "red")
```



Assessing Model Performance

- We can also look at the coverage of the predictive densities.

```
lower95 <- qlnorm(0.025, meanlog = mu, sdlog = sigma)
upper95 <- qlnorm(0.975, meanlog = mu, sdlog = sigma)
coverage95 <- sum(testData_Y > lower95 &
                  testData_Y < upper95)/length(testData_Y)
print(coverage95)
```

```
## [1] 0.9523337
```

Assessing Model Performance

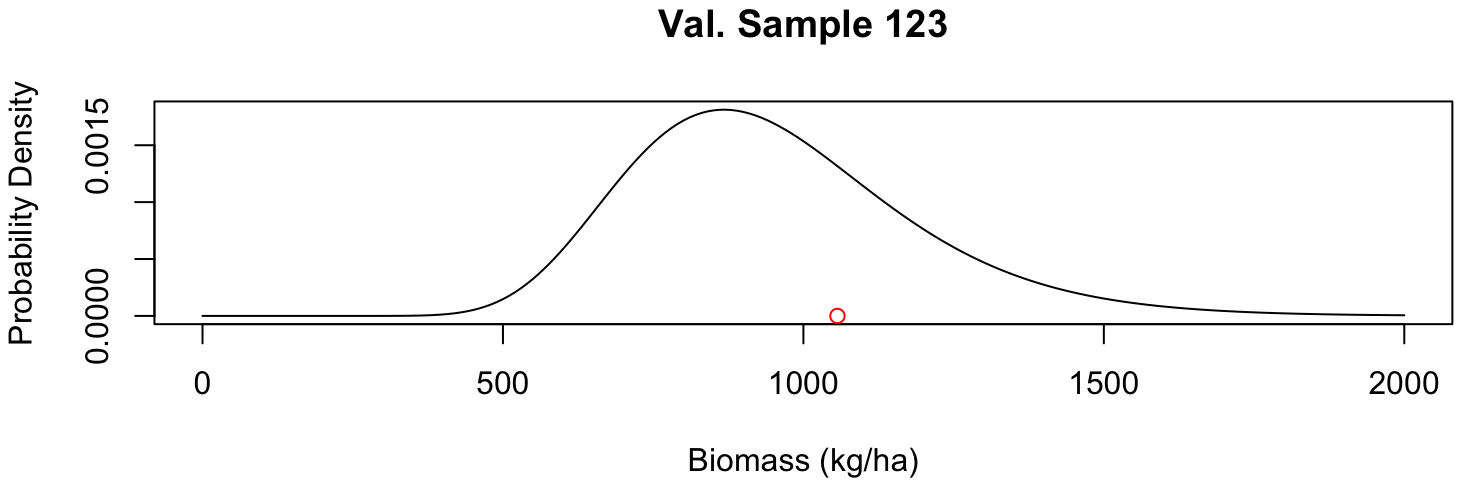
- We can also look at the coverage of the predictive densities.

```
lower50 <- qlnorm(0.25, meanlog = mu, sdlog = sigma)
upper50 <- qlnorm(0.75, meanlog = mu, sdlog = sigma)
coverage50 <- sum(testData_Y > lower50 &
                  testData_Y < upper50)/length(testData_Y)
print(coverage50)
```

```
## [1] 0.6047666
```

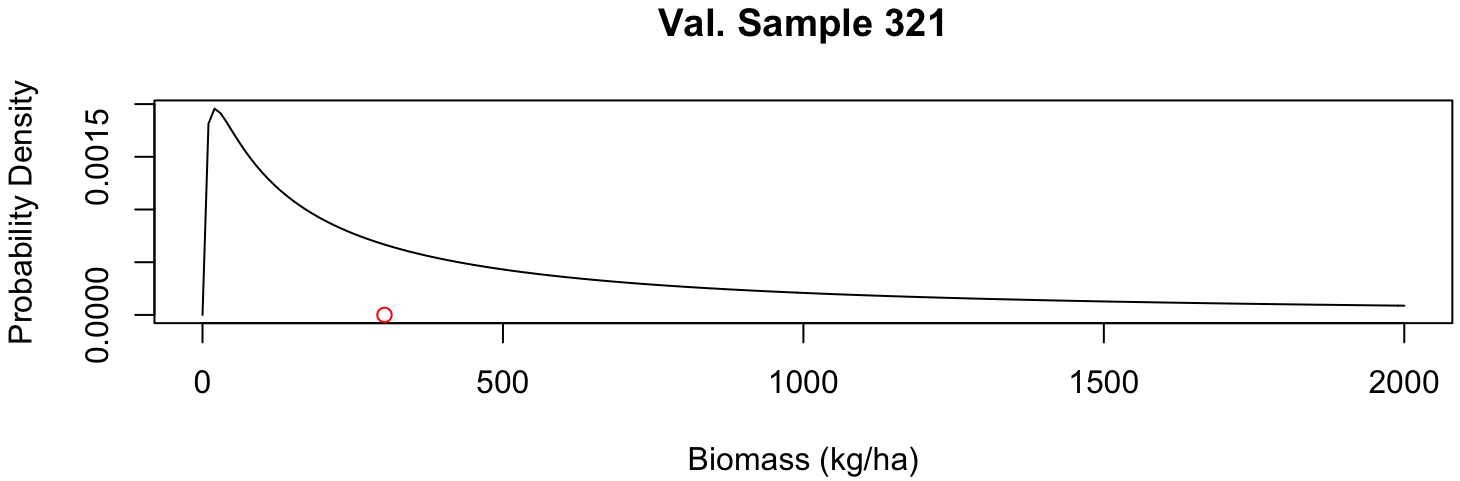
Assessing Model Performance

```
xgrid <- seq(0, 2000, 10)
plotIndex = 123
yieldDist <- dlnorm(xgrid, meanlog = mu[plotIndex],
                    sdlog = sigma[plotIndex])
plot(xgrid, yieldDist, ty = "l",
     main = paste0("Val. Sample ", plotIndex), xlab = "Biomass (kg/ha)",
     ylab = "Probability Density")
points(x = testData_Y[plotIndex], y = 0, col = "red")
```



Assessing Model Performance

```
plotIndex = 321
yieldDist <- dlnorm(xgrid, meanlog = mu[plotIndex],
                    sdlog = sigma[plotIndex])
plot(xgrid, yieldDist, ty = "l",
     main = paste0("Val. Sample ", plotIndex), xlab = "Biomass (kg/ha)",
     ylab = "Probability Density")
points(x = testData_Y[plotIndex], y = 0, col = "red")
```



Some things to try

- How do your results change if you include more nodes in the hidden layers?
- How do your results change if you make the network deeper?
- How could you modify the loss function to work with the zero wheat yield values that we removed?
- Try including some form of regularisation (e.g. Dropout or L1 regularisation) in the hidden layers to help prevent overfitting.