Single-Layer NN Notes

Daniel Polites

(ISLR mainly)

Model Form

We have an input vector of p variables $X = \{x_1, x_2, \ldots, x_p\}$, and an output scalar Y. We want to build a function $f : \mathbb{R}^p \to \mathbb{R}$ to approximate Y.

For a single layer NN, we have an input layer, hidden layer (with K activations), and output layer. Thus, the model's form is:

$$\begin{aligned} f(X) &= \beta_0 + \sum_{k=1}^{K} [\beta_k * h_k(X)] \\ &= \beta_0 + \sum_{k=1}^{K} \left[\beta_k * g \left(w_{k0} + \sum_{j=1}^{p} [w_{kj} * X_j] \right) \right] \end{aligned}$$

we have k indexing our hidden layer neurons, j indexing the weights within each neuron as they relate to each input variable $\{1, 2, \ldots, p\}$. $g(\cdot)$ is our activation function.

This model form is built in 2 steps:

$h_k(X)$ is known as the activation of the kth neuron of the hidden layer; it is denoted A_k:

$$A_k = h_k(X) = g \left(w_{k0} + \sum_{j=1}^{p} [w_{kj} * X_j] \right)$$

These get fed into the output layer, so that:

$$f(X) = \beta_0 + \sum_{k=1}^{K} (\beta_k * A_k)$$

Activation Functions

(add derivatives)

Sigmoid:

$$g(z) = \frac{e^z}{1+e^z} = \frac{1}{1+e^{-z}}$$

ReLU:

$$g(z) = (z)_+ = \begin{cases} 0 & \text{if } z < 0 \\ z & \text{otherwise} \end{cases}$$

softmax:

$$f_s(X) = P(Y = s|X) = \frac{e^{Z_s}}{\sum_{l=1}^{w} e^{Z_l}}$$

^^ used for the output layer of a categorical response network.

Loss Functions

For a quantitative response variable, typical to use a squared-error loss function:

$$\sum_{i=1}^{n} \left[(y_i - f(x_i))^2\right]$$

For a qualitative / categorical response variable, typical to use cross-entropy:

$$-\sum_{i=1}^{n}\sum_{m=1}^{w} [y_{im} * \ln(f_m(x_i))]$$

Where w is the number of output categories. The behavior of this function is such that if the correct category is predicted as 1, the loss is 0. Otherwise, higher certainty for the correct category is rewarded for the correct answer, and lower certainty is punished.

The output matrix Y has been transformed using one-hot encoding in this circumstance, that's how there are multiple output dimensions (details).

Recall that y_{im} can only be 1 for the correct category; otherwise it is 0. So for each observation, only adding one number here to the total loss.

(3B1B also shows the sum of squared loss for the probability of each category)

Parameterization

For a single-layer neural network, we have 2 parameter matrices; one for the weights of the hidden layer, and one for the weights of the output layer. These are denoted **W** and **B**, respectively.

In **W**, each row represents an input (with the first row being the '1' input / the neuron's 'bias'); each column represents a neuron:

$$\mathbf{W} = \begin{bmatrix} w_{1,0} & w_{2,0} & \cdots & w_{K,0} \\ w_{1,1} & w_{2,1} & \cdots & w_{K,1} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1,p} & w_{2,p} & \cdots & w_{K,p} \end{bmatrix}$$

For $\mathbf{B}$, each row is a hidden-layer neuron's activation (& a bias term).

If the output is quantitative, there is only 1 column for the output:

$$\mathbf{B} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

If the output is qualitative, there is one column per output category:

$$\mathbf{B} = \begin{bmatrix} \beta_{1,0} & \beta_{2,0} & \cdots & \beta_{w,0} \\ \beta_{1,1} & \beta_{2,1} & \cdots & \beta_{w,1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{1,K} & \beta_{2,K} & \cdots & \beta_{w,K} \end{bmatrix}$$

We can combine $\mathbf{W}$ and $\mathbf{B}$ into one parameter vector:

$$\theta = \begin{bmatrix} w_{1,0} \\ w_{2,0} \\ \vdots \\ w_{K,p} \\ \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

Note that $\mathbf{W}$ is a $(p+1) \times K$ dimension matrix, and $\mathbf{B}$ is a $(K+1) \times w$ dimension matrix. So, θ has $(p+1) * K + (K+1) * w$ total parameters.

Network Fitting

Starting with a quantitative output. Our goal is to find:

$$\arg\min_{\theta} \sum_{i=1}^{n} \mathcal{L}(y_i, f(x_i))$$

We will use a scaled squared-error loss function:

$$\sum_{i=1}^{n} \frac{1}{2} \left[(y_i - f(x_i))^2 \right]$$

The scaling make for easier derivative-taking down the line. Recall that:

$$f(x_i) = \beta_0 + \sum_{k=1}^{K} \left[\beta_k * g \left(w_{k0} + \sum_{j=1}^{p} [w_{kj} * x_{ij}] \right) \right]$$

So, we are trying to find:

$$\arg\min_{\theta} \sum_{i=1}^{n} \frac{1}{2} \left[y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) \right]^2$$

We will denote the summation (our objective function) $\mathcal{C}(\theta)$.

This is nearly impossible to calculate by taking the derivative with respect to every variable and solving for a simultaneous 0; however, we can approximate solutions via gradient descent.

Gradient Descent

Our goal is to find $\arg\min_\theta \mathcal{C}(\theta)$ with gradient descent:

1. Start with a guess θ^0 for all parameters in θ, and set $t = 0$
2. Iterate until $\mathcal{C}(\theta)$ fails to decrease:
   - $\theta^{t+1} \leftarrow \theta^t - \rho * \nabla \mathcal{C}(\theta)$

ρ is our learning rate: it controls how quickly we respond to the gradient. $\nabla \mathcal{C}(\theta)$ points in the direction of the greatest increase, so we subtract it to move in the direction of the greatest decrease. Our change in parameter values is proportional to both the learning rate and the gradient magnitude.

The last step for us is taking the gradient. In our parameter vector, we have two 'types' of parameters: those that came from $\mathbf{W}$, and those that came from $\mathbf{B}$. These can be split further into those which are intercept terms (—> simpler derivatives) or not.

We will start by manipulating the notation of our objective function to make it easier to work with:

- let $z_{ik} = w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}$
  - so z_{ik} is the ith input of the activation function of the kth hidden-layer neuron
- let $\hat{y}_i = \beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik})$
  - so $\hat{y}_i$ is our ith prediction
- let $\hat{\epsilon}_i = \hat{y}_i - y_i$
  - so $\hat{\epsilon}_i$ is our ith residual
  - note that $\hat{\epsilon}_i = \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i$
  - (against convention here because this is a negative residual; playing fast & loose w/ notation)
- because $(a-b)^2 = (b-a)^2$, we will flip y and $\hat{y}$ in our objective function

So we have:

$$\mathcal{C}(\theta) = \sum_{i=1}^{n} \frac{1}{2} \left[y_i - \left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\hat{y}_i - y_i \right]^2$$

$$= \sum_{i=1}^{n} \frac{1}{2} \left[\hat{\epsilon}_i \right]^2$$

Taking our derivatives:

Beta: Intercept

$$\frac{\partial \mathcal{C}}{\partial \beta_0} = \frac{\partial}{\partial \beta_0} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]$$

$$= \sum_{i=1}^{n} \hat{\epsilon}_i$$

Beta: Coefficients

$$\frac{\partial \mathcal{C}}{\partial \beta_k} = \frac{\partial}{\partial \beta_k} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial \beta_k} [\beta_k * g(z_{ik})]$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] g(z_{ik})$$

$$= \sum_{i=1}^{n} \hat{\epsilon}_i \; g(z_{ik})$$

W: Intercepts

$$\frac{\partial \mathcal{C}}{\partial w_{k0}} = \frac{\partial}{\partial w_{k0}} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial w_{k0}} [\beta_k * g(z_{ik})]$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \; \frac{\partial}{\partial w_{k0}} g(z_{ik})$$

$$= \sum_{i=1}^{n} \hat{\epsilon}_i \; \beta_k \; g'(z_{ik})$$

note that $\frac{\partial}{\partial w_{k0}} z_{ik} = \frac{\partial}{\partial w_{k0}} \left[w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right] = 1$

W: Coefficients

$$\frac{\partial \mathcal{C}}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}} \sum_{i=1}^{n} \frac{1}{2} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right]^2$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \frac{\partial}{\partial w_{kj}} [\beta_k * g(z_{ik})]$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ \frac{\partial}{\partial w_{kj}} g(z_{ik})$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ g'(z_{ik}) \ \frac{\partial}{\partial w_{kj}} z_{ik}$$

$$= \sum_{i=1}^{n} \left[\left(\beta_0 + \sum_{k=1}^{K} \beta_k * g(z_{ik}) \right) - y_i \right] \beta_k \ g'(z_{ik}) \ x_{ij}$$

$$= \sum_{i=1}^{n} \hat{\epsilon}_i \ \beta_k \ g'(z_{ik}) \ x_{ij}$$

note that $\frac{\partial}{\partial w_{kj}} z_{ik} = \frac{\partial}{\partial w_{kj}} \left[w_{k0} + \sum_{j=1}^{p} w_{kj} x_{ij} \right] = x_{ij}$

Combining

Given:

$$\theta = \begin{bmatrix} w_{1,0} \\ w_{2,0} \\ \vdots \\ w_{K,p} \\ \beta_0 \\ \beta_1 \\ \vdots \\ \beta_K \end{bmatrix}$$

and

$$\mathcal{C}(\theta) = \sum_{i=1}^{n} \frac{1}{2} \left[\hat{\epsilon}_i \right]^2$$

We have computed:

$$\nabla \mathcal{C}(\theta) = \begin{bmatrix} \frac{\partial \mathcal{C}}{\partial w_{1,0}} \\ \frac{\partial \mathcal{C}}{\partial w_{2,0}} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial w_{1,1}} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial w_{K,p}} \\ \frac{\partial \mathcal{C}}{\partial \beta_0} \\ \frac{\partial \mathcal{C}}{\partial \beta_1} \\ \vdots \\ \frac{\partial \mathcal{C}}{\partial \beta_K} \end{bmatrix} = \sum_{i=1}^{n} \begin{bmatrix} \hat{\epsilon}_i \ \beta_1 \ g'(z_{i1}) \\ \hat{\epsilon}_i \ \beta_2 \ g'(z_{i2}) \\ \vdots \\ \hat{\epsilon}_i \ \beta_1 \ g'(z_{i1}) \ x_{i1} \\ \vdots \\ \hat{\epsilon}_i \ \beta_K \ g'(z_{ik}) \ x_{ip} \\ \hat{\epsilon}_i \\ \hat{\epsilon}_i \ g(z_{i1}) \\ \vdots \\ \hat{\epsilon}_i \ g(z_{ik}) \end{bmatrix}$$

Example

A simple example of using a small single-layer neural network to act on simulated data:

Generate Data

For now, having 3 inputs and combining them to create y, with a random error term. Would like to tweak the setup eventually.

```
## create data:
n <- 1000
p <- 3

# initialize Xs
X <- data.frame(X1 = runif(n = n, min = -10, max = 10),
                X2 = rnorm(n = n, mean = 0, sd = 10),
                X3 = rexp(n = n, rate = 1)) %>%
  as.matrix(nrow = n,
            ncol = p)

# get response
Y <- X[, 1] + 10 * sin(X[, 2])^2 + 10 * X[, 3] + rnorm(n = 1000)
```

Parameter Setup

We will have 2 hidden-layer neurons and a single quantitative output, so $\mathbf{W}$ will be 4×2 and $\mathbf{B}$ will be 3×1:

```
## NN properties
K <- 2

## initialize parameter matrices
W <- matrix(data = runif(n = (p + 1) * K, min = -1, max = 1),
            nrow = (p + 1),
            ncol = K)

B <- matrix(data = runif(n = (K + 1), min = -1, max = 1),
```

```
            nrow = (K + 1),
            ncol = 1)

## Specify Link Functions & Derivatives:
# identity
# g <- function(x) {x}
# g_prime <- function(x) {1}

# sigmoid
g <- function(x) {1 / (1 + exp(-x))}
g_prime <- function(x) {exp(-x) / (1 + exp(-x))^2}

# ReLU
# g <- function(x) {if (x < 0) {0} else {x}}
# g_prime <- function(x) {if (x < 0) {0} else {1}}
```

Output

How the NN will calculate the output:

```
## create output function
NN_output <- function(X, W, B) {
  cbind(1, g(cbind(1, X) %*% W)) %*% B
}

example <- NN_output(X = X,
                     W = W,
                     B = B)

example[1:5]
```

```
## [1] -0.09605139 -0.16496338 -0.11193759 -0.22617287 -0.22526120
```

Gradient Descent

for now, looping through each observation's gradient then taking the sum — much slower than using matrix/arrays, which will eventually happen:

```
GD_iteration <- function(X, Y, W, B, rho = 1) {

  ## get errors
  errors <- NN_output(X = X, W = W, B = B) - Y

  ## all at once (do eventually)
  # errors_W <- array(data = rep(errors, each = (p + 1) * K),
  #                   dim = c(p + 1, K, n))
  #
  # X_W <- array(data = t(cbind(1, X))[ , rep(1:n, each = 3)],
  #              dim = c(p + 1, K, n))
  #
  # B_W <- array(data = t(B[-1, rep(1, (p + 1))]),
```

```
#               dim = c(p + 1, K, n))

## get each obs' gradient
gradient_array_W <- array(dim = c((p + 1), K, nrow(X)))
gradient_array_B <- array(dim = c((K + 1), 1, nrow(X)))

for (i in 1:nrow(X)) {

  ## W
  errors_W <-  matrix(errors[i],
                      nrow = (p + 1),
                      ncol = K)

  B_W <- matrix(B[-1, ],
                nrow = (p + 1),
                ncol = K,
                byrow = TRUE)

  X_W <- matrix(c(1, X[i, ]),
                nrow = (p + 1),
                ncol = K,
                byrow = FALSE)

  g_prime_z_W <- apply(X = c(1, X[i, ]) %*% W,
                       MARGIN = 2,
                       FUN = g_prime) %>%
    matrix(nrow = (p + 1),
           ncol = K,
           byrow = FALSE)

  del_W <- errors_W * B_W * g_prime_z_W * X_W

  gradient_array_W[ , , i] <- del_W

  ## B
  errors_B <- matrix(errors[i],
                     nrow = (K + 1),
                     ncol = 1)

  g_z_B <- apply(X = c(1, X[i, ]) %*% W,
                 MARGIN = 2,
                 FUN = g) %>%
    c(1, .) %>%
    matrix(nrow = (K + 1),
           ncol = 1)

  del_B <- errors_B * g_z_B

  gradient_array_B[ , , i] <- del_B
}

## get gradients
del_W_all <- apply(X = gradient_array_W,
```

```
                      MARGIN = c(1, 2),
                      FUN = mean)

  del_B_all <- apply(X = gradient_array_B,
                     MARGIN = c(1, 2),
                     FUN = mean)

  ## perform iteration
  W_out <- W - rho * del_W_all
  B_out <- B - rho * del_B_all

  ## return
  return(list(W = W_out,
              B = B_out))
}

## test run
iteration <- GD_iteration(X = X,
                          Y = Y,
                          W = W,
                          B = B,
                          rho = 1 / 100)

## in loss:
sum((NN_output(X = X, W = W, B = B) - Y)^2)
```

```
## [1] 363335.7
```

```
## out loss:
sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)
```

```
## [1] 357050.9
```

Iterate

Employ gradient descent until objective function stops decreasing:

```
threshold <- 1000

done_decreasing <- FALSE

iteration <- list()
iterations <- list()

iteration$W <- W
iteration$B <- B

iter <- 1

initial_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)
```

```r
while (!done_decreasing) {
  ## get input loss
  in_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

  ## perform iteration
  iteration <- GD_iteration(X = X,
                            Y = Y,
                            W = iteration$W,
                            B = iteration$B,
                            rho = 1 / 100)

  ## get output loss
  out_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

  ## evaluate
  if (abs(in_objective - out_objective) < threshold) {
    done_decreasing <- TRUE
  }

  # print(iter)
  # print(out_objective)

  iterations[[iter]] <- cbind(matrix(iteration$W, nrow = 1),
                              matrix(iteration$B, nrow = 1))

  iter <- iter + 1
}

final_objective <- sum((NN_output(X = X, W = iteration$W, B = iteration$B) - Y)^2)

## number of iterations
iter <- iter - 1
iter
```

```
## [1] 72
```

```r
## loss improvement ratio
initial_objective
```

```
## [1] 363335.7
```

```r
final_objective
```

```
## [1] 116370.9
```

```r
final_objective / initial_objective
```

```
## [1] 0.3202848
```

```
## input W
W
```

```
##            [,1]        [,2]
## [1,]  0.4037782 -0.50135717
## [2,] -0.4417944 -0.39889765
## [3,]  0.8015552  0.63845282
## [4,]  0.3183736 -0.02490377
```

```
## output W
iteration$W
```

```
##           [,1]       [,2]
## [1,] 0.6798924 -0.2368236
## [2,] 0.3820039  0.3896985
## [3,] 0.0293259 -0.0697037
## [4,] 1.9008662  1.4390933
```

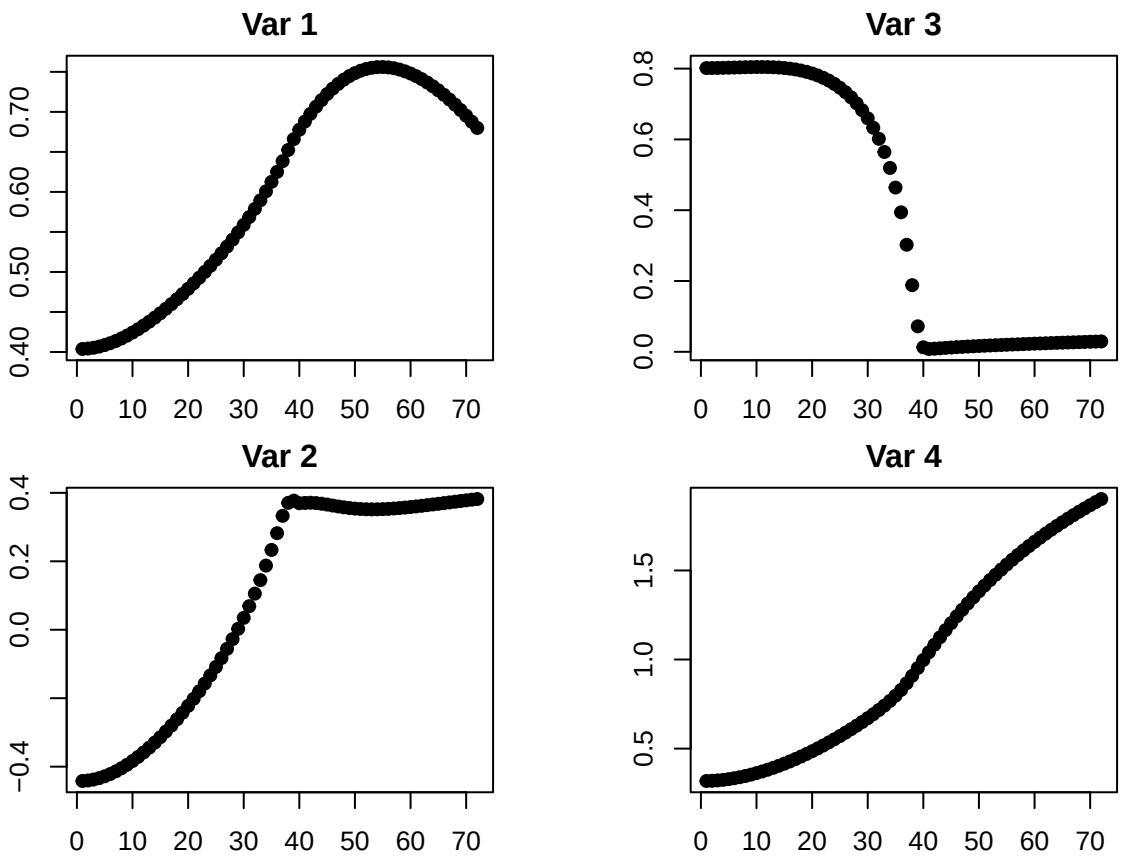
```
## input B
B
```

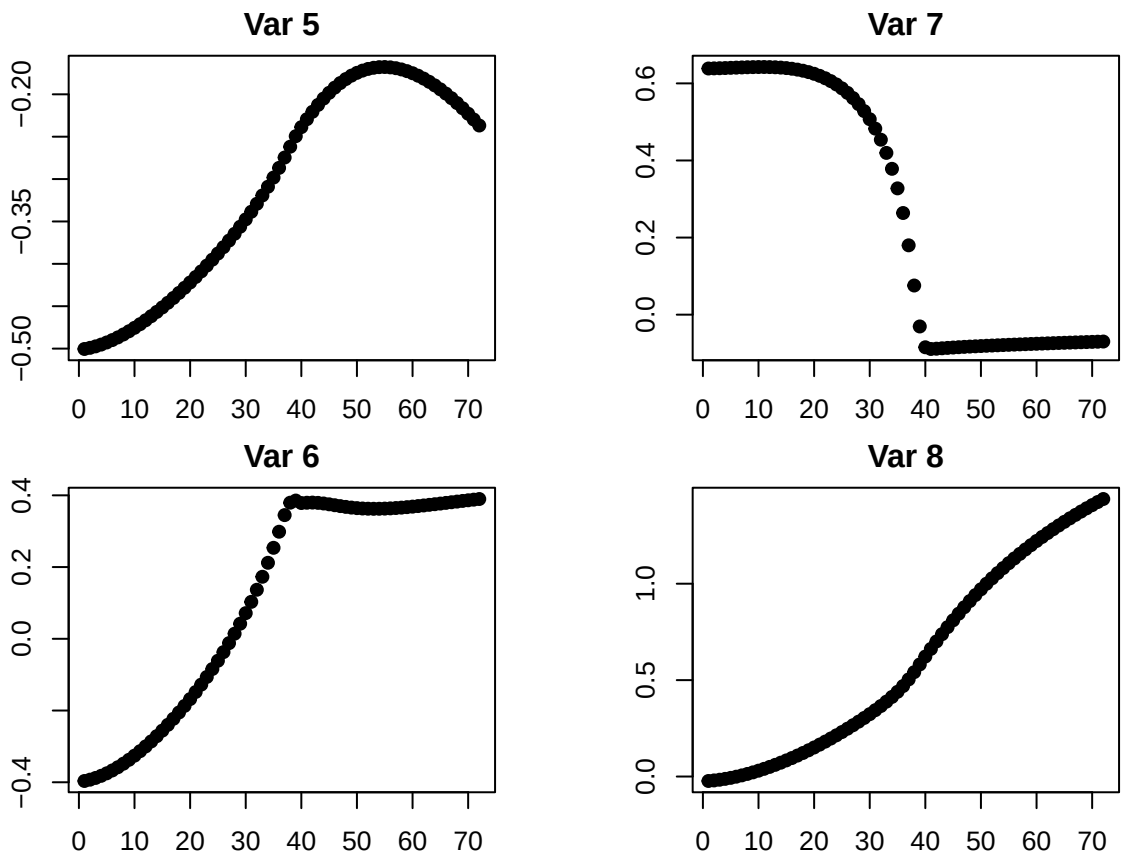
```
##               [,1]
## [1,] -0.2261729394
## [2,] -0.0007045274
## [3,]  0.1314884122
```

```
## output B
iteration$B
```

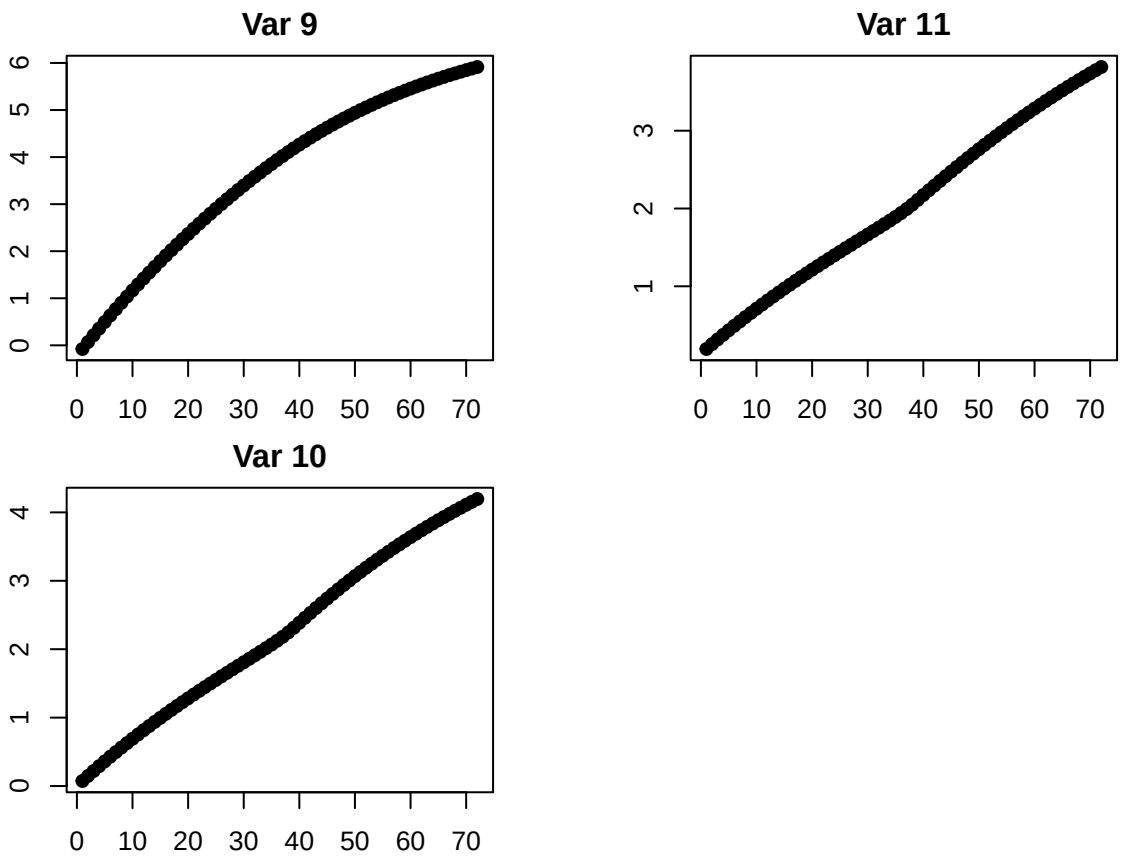
```
##          [,1]
## [1,] 5.913779
## [2,] 4.195002
## [3,] 3.818744
```

```
## plots
iterations <- do.call(rbind, iterations)

par(mfcol = c(2, 2))
par(mar = c(2, 4.1, 2, 2.1))

for (i in 1:ncol(iterations)) {
  plot(x = 1:iter,
       y = iterations[ , i],
       pch = 19,
       main = paste("Var", i),
       ylab = "",
       xlab = "")
}
```


Var 1
0.40
0.50
0.60
0.70
0
10
20
30
40
50
60
70
Var 3
0.0
0.2
0.4
0.6
0.8
0
10
20
30
40
50
60
70
Var 2
−0.4
0.0
0.2
0.4
0
10
20
30
40
50
60
70
Var 4
0.5
1.0
1.5
0
10
20
30
40
50
60
70




```
par(mfcol = c(1, 1))
```




```
par(mar = c(5.1, 4.1, 4.1, 2.1))
```

vis concept for the future: create a matrix of plots the same layout as W, B

Indices & Matrix Notation