

## Lista 2

José Felipe

- a) Altere seu código da Lista 1 (ou, se preferir, os códigos disponibilizados como gabarito) para implementar a técnica *dropout* na camada de entrada e na camada intermediária. Use  $p = 0,6$ , onde  $p$  representa a probabilidade de inclusão de cada neurônio. Reporte o custo dessa rede.

```
# Função para realizar o forward propagation, dados theta e x, com dropout
forward_prop_dropout<-function(theta, x, p = 0.6) {
  # Transformação de x para um formato de matriz
  ifelse(is.double(x), x<-as.matrix(x), x<-t(as.matrix(x)))
  # Extração dos parâmetros
  W1<-matrix(data = theta[1:4], nrow = 2)
  W2<-matrix(data = theta[5:6], nrow = 2)
  b1<-theta[7:8]
  b2<-theta[9]
  # Camada de entrada com dropout
  mask_input<-rbinom(length(x), 1, p)
  x_input<-x * mask_input
  # Camada escondida com dropout
  mask_hidden<-rbinom(ncol(x_input), 1, p)
  a<-matrix(data = rep(b1, ncol(x_input)), nrow = 2) + W1 %*% x_input
  h<-sigmoide(a) * mask_hidden
  # Previsão
  y_hat<-as.double(b2 + t(W2) %*% h)

  return(y_hat)
}

# Aplicando a função com dropout para os valores de theta
forward_prop_dropout(theta = rep(.1, 9), x = c(1, 1))
```

```
## [1] 0.2148885
```

```
# Função para realizar o back-propagation com dropout
back_prop_dropout<-function(theta, x, y, p = 0.6) {
  ### Primeiro, deve-se realizar o forward propagation com dropout
  ifelse(is.double(x), x<-as.matrix(x), x<-t(as.matrix(x)))
  W1<-matrix(data = theta[1:4], nrow = 2)
  W2<-matrix(data = theta[5:6], nrow = 2)
  b1<-theta[7:8]
  b2<-theta[9]

  mask_input<-rbinom(length(x), 1, p)
  x_input<-x * mask_input
```

```

mask_hidden<-rbinom(ncol(x_input), 1, p)
a<-matrix(data = rep(b1, ncol(x_input)), nrow = 2) + W1 %*% x_input
h<-sigmoide(a) * mask_hidden
y_hat<-as.double(b2 + t(W2) %*% h)

### Em seguida, passamos para a implementação do back propagation com dropout
## Camada final: k = 2
g<-2*(y - y_hat)/length(y)
grad_b2<-sum(g)
grad_W2<-g %*% t(h)
g<-W2 %*% g * derivada_sigmoide(a) * mask_hidden

## Camada escondida: k = 1
grad_b1<-rowSums(g * mask_input)
grad_W1<-g %*% t(x)
g<-W1 %*% g

vetor_grad<-c(grad_W1, grad_W2, grad_b1, grad_b2)
names(vetor_grad)<-c(paste0("w", 1:6), paste0("b", 1:3))

return(vetor_grad)
}

# Aplicando o back-propagation com dropout no banco de treinamento e theta especificado no enunciado
back_prop_dropout(theta = rep(.1, 9), x = x_treino, y = y_treino)

##          w1          w2          w3          w4          w5          w6
## -0.1021515 -0.1022425  0.3877903  0.3850143 -13.5367943 -13.5496552
##          b1          b2          b3
## -0.3861894 -0.3861052 -43.4947737

# Lista para receber os parâmetros estimados em cada iteração com dropout
theta_est_dropout<-list()
# Theta inicial com dropout
theta_est_dropout[[1]]<-rep(0, 9)

# Vetores para receber os custos de treino e teste em cada iteração com dropout
custo_treino_dropout<-custo_teste_dropout<-numeric(M)

# Execução com dropout
for(i in 1:M) {
  grad<-back_prop_dropout(theta = theta_est_dropout[[i]], x = x_treino, y = y_treino)
  custo_treino_dropout[i]<-mse_cost(y_treino, forward_prop_dropout(theta_est_dropout[[i]], x_treino))
  custo_teste_dropout[i]<-mse_cost(y_teste, forward_prop_dropout(theta_est_dropout[[i]], x_teste))

  theta_est_dropout[[i+1]]<-theta_est_dropout[[i]] - epsilon * grad
}

# Custo da rede com dropout
custo_rede_dropout<-custo_teste_dropout[which.min(custo_teste_dropout)]
round(custo_rede_dropout, 3)

## [1] 170.481

```

b) Considerando os pesos obtidos em a), para a primeira observação do conjunto de teste, gere 200 previsões ( $\hat{y}_{1,1}, \dots, \hat{y}_{1,200}$ ), uma para cada sub-rede amostrada aleatoriamente. Use as previsões para construir uma estimativa pontual e um intervalo de confiança para  $y_1$ . Veja a Figura 7.6 do livro *Deep Learning*. Note que com esse procedimento, não é preciso assumir normalidade para os erros, como fizemos na Lista 1.

```
set.seed(1) # Para reproduzibilidade dos resultados

# Gerando 200 previsões com dropout para a primeira observação do conjunto de teste
n_subredes<-200
previseoes<-replicate(n_subredes, forward_prop_dropout(theta = min_theta, x = x_teste[1, ]))

# Estimativa pontual para y1
estimativa_pontual<-mean(previseoes)
estimativa_pontual

## [1] 20.58615

# Intervalo de confiança para y1
intervalo_confianca<-quantile(previseoes, c(0.025, 0.975))
intervalo_confianca

##      2.5%    97.5%
## 11.37075 27.24904
```

c) Repita o item b) para gerar estimativas pontuais para cada observação do conjunto de testes.

```
library(doParallel)

## Carregando pacotes exigidos: foreach

##
## Attaching package: 'foreach'

## The following objects are masked from 'package:purrr':
##
##     accumulate, when

## Carregando pacotes exigidos: iterators

## Carregando pacotes exigidos: parallel

library(foreach)

# Definir o número de núcleos a serem usados
n_cores<-detectCores()-1

# Configurar o cluster paralelo
cl<-makeCluster(n_cores)
registerDoParallel(cl)
```

```

# Vetor para armazenar as estimativas pontuais para cada observação do conjunto de teste
estimativas_pontuais<-foreach(i = 1:nrow(x_teste), .combine = c) %dopar% {
  previsoes<-replicate(n_subredes, forward_prop_dropout(theta = min_theta, x = x_teste[i, ]))
  mean(previsoes)
}

# Encerrar o cluster paralelo
stopCluster(cl)

estimativas_pontuais[1:10]

```

```

## [1] 21.52923 20.26649 19.55637 14.37576 14.91666 14.53575 17.59911 20.38319
## [9] 20.18592 18.46414

```

- d) Use a regra *weight scaling inference rule* (página 263 do livro *Deep Learning*) para gerar novas estimativas para as observações do conjunto de testes. Qual dos procedimentos (o do item c) ou o utilizado neste item) produziu melhores resultados? Considerando o tempo computacional de cada um, qual você escolheria nessa aplicação?

```

# Aplicando a regra weight scaling inference
estimativas_weight_scaling<-estimativas_pontuais * min_theta[9]

# Calculando as novas estimativas do conjunto de testes
novas_estimativas<-forward_prop(theta = min_theta, x = x_teste)

# Calculando o custo das novas estimativas
custo_weight_scaling<-mse_cost(y_teste, novas_estimativas)
round(custo_weight_scaling, 3)

```

```

## [1] 142.541

```

Em relação ao tempo computacional, o item c) é mais rápido, pois não requer a multiplicação pelos pesos originais e o cálculo do custo é direto.

## 2

- a) Ajuste a rede neural especificada na Lista 1 usando o *Keras*. Compare com sua implementação (Lista 1, item e) quanto ao tempo computacional e ao custo obtido no conjunto de teste. Use o mesmo algoritmo de otimização (*full gradient descent*) e ponto de partida.

```

pacman::p_load(keras, tensorflow, tidyverse)
# Definir a arquitetura da rede neural
model<-keras_model_sequential() %>%
  layer_dense(units = 2, activation = "sigmoid", input_shape = c(2)) %>%
  layer_dense(units = 1, activation = "linear")

# Compilar o modelo
model %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_sgd(learning_rate = 0.01)
)

```

```

# Treinar o modelo
history<-model %>% fit(
  x = as.matrix(x_treino),
  y = y_treino,
  epochs = 40,
  batch_size = 60,
  verbose = 0
)

# Calcular o custo no conjunto de teste
y_hat_keras<-model %>% predict(as.matrix(x_teste))
custo_keras<-mse_cost(y_teste, y_hat_keras)
round(custo_keras, 3)

## [1] 94.426

```

b) Ajuste a rede neural mais precisa (medida pelo MSE calculado sobre o conjunto de validação) que conseguir, com a arquitetura que quiser. Use todos os artifícios de regularização que desejar (*weight decay, Bagging, dropout, Early stopping*). Reporte a precisão obtida.

```

# Definir a arquitetura da rede neural
model<-keras_model_sequential() %>%
  layer_dense(units = 128, activation = "relu", input_shape = c(2)) %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = "linear")

# Compilar o modelo
model %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(),
  metrics = c("accuracy")
)

# Treinar o modelo
history<-model %>% fit(
  x = as.matrix(x_treino),
  y = y_treino,
  epochs = 40,
  batch_size = 790,
  validation_split = 0.0000001,
  verbose = 0
)

# Calcular a precisão no conjunto de validação

```

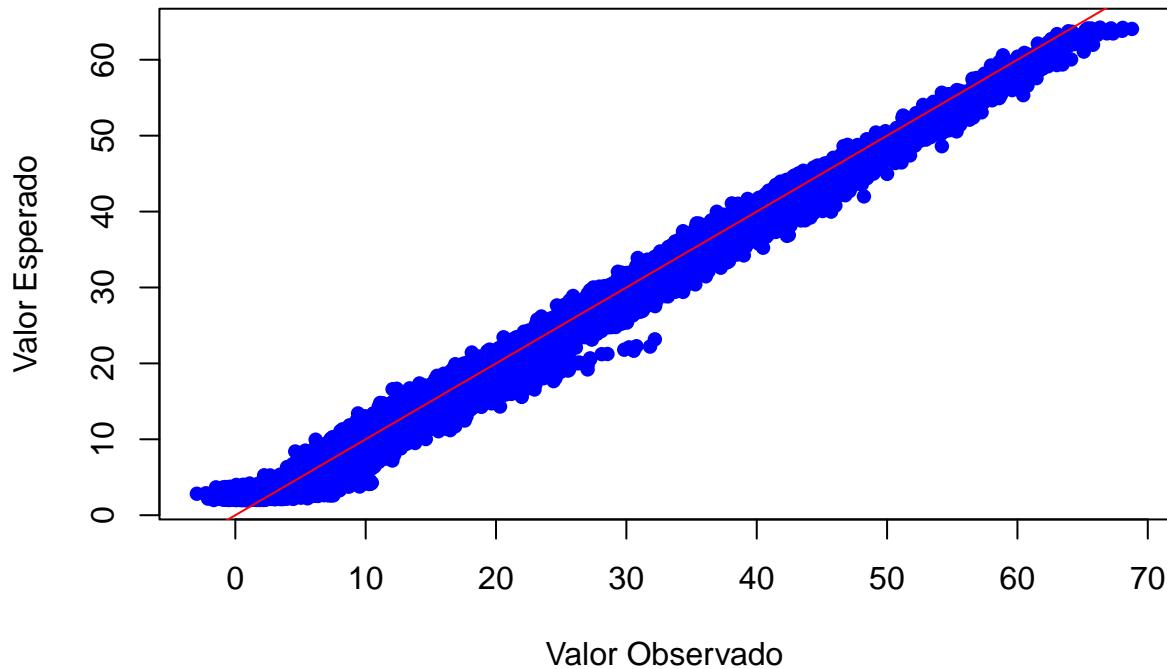
```
y_pred<-model %>% predict(as.matrix(x_treino))
mse<-mse_cost(y_treino, y_pred)
mse
```

```
## [1] 2.435322
```

c)

```
# Previsões no conjunto de teste
y_pred_test<-model %>% predict(as.matrix(x_teste))

# Gráfico do valor observado em função do valor esperado
plot(y_teste, y_pred_test, pch = 16, col = "blue", xlab = "Valor Observado", ylab = "Valor Esperado")
abline(0, 1, col = "red")
```



No gráfico observado, podemos notar que a função de previsão do modelo tem um desempenho significativamente melhor em comparação com o modelo da lista 1. A linha vermelha representa o resultado esperado em um cenário ideal, onde os valores observados e esperados seriam perfeitamente iguais.

No entanto, mesmo com um modelo bem ajustado, ainda há uma pequena variação nos valores observados em relação aos valores esperados. Essa variação é representada pelos pontos dispersos em torno da linha vermelha. Essa dispersão está relacionada ao desvio padrão da variável Y, que é igual a 1, conforme a distribuição normal.

d) Use a função de previsão do *Keras* para prever o valor da variável resposta  $\hat{y} = f(x_1 = 1, x_2 = 1; \theta)$ , para  $\theta$  definido de acordo com a rede ajustada. (Veja o item a) da Lista 1).

```
(valor_predito<-predict(model, matrix(c(1, 1), nc = 2)))
```

```
##          [,1]
## [1,] 14.7888
```

e) Neste exemplo meramente didático, conhecemos a superfície que estamos estimando. Apresente, lado a lado, a Figura 1 da Lista 1 e a superfície estimada pela sua rede neural. Para tanto, basta trocar a variável `mu` pelos valores preditos pela rede. Comente os resultados.

```
x1_mesh<-seq(-2, 2, length.out = 1000)
x2_mesh<-seq(-2, 2, length.out = 1000)
grid<-expand.grid(x1 = x1_mesh, x2 = x2_mesh)

# Prever os valores para a malha usando a rede neural
y_pred_mesh<-model %>% predict(as.matrix(grid))
figura1<-ggplot(data = dados, aes(x = x1.obs, y = x2.obs)) +
  geom_tile(aes(fill = y), width = 0.1, height = 0.1) +
  scale_fill_gradientn(colors = c("#FFFFDD", "#3182BD")) +
  labs(title = "Superfície Original da Lista 1", x = "x1", y = "x2", fill = "y") +
  theme_minimal()

# Criar uma nova figura com os valores preditos pela rede neural
figura2<-ggplot(data = grid, aes(x = x1, y = x2)) +
  geom_tile(aes(fill = y_pred_mesh), width = 0.1, height = 0.1) +
  scale_fill_gradientn(colors = c("#FFFFDD", "#3182BD")) +
  labs(title = "Superfície Estimada pela Rede Neural", x = "x1", y = "x2", fill = "y") +
  theme_minimal()

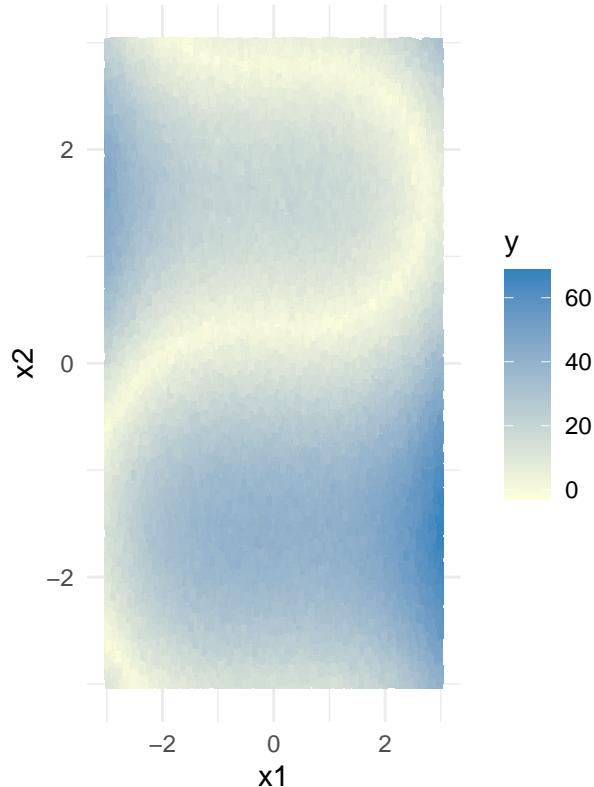
# Exibir as figuras lado a lado
library(gridExtra)

## 
## Attaching package: 'gridExtra'

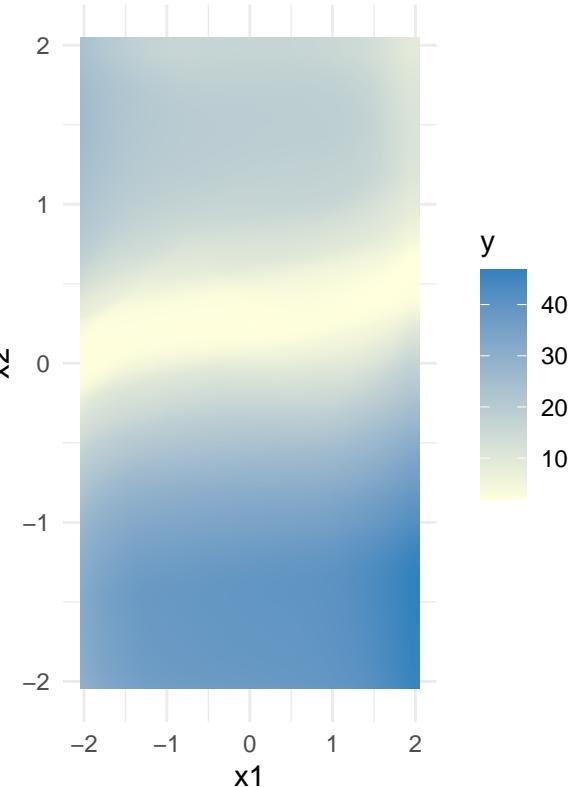
## The following object is masked from 'package:dplyr':
## 
##     combine

# Exibir as figuras lado a lado
figura_comparativa<-grid.arrange(figura1, figura2, ncol = 2)
```

Superfície Original da Lista 1



Superfície Estimada pela Rede Ne



```
# Exibir a figura comparativa
figura_comparativa
```

```
## TableGrob (1 x 2) "arrange": 2 grobs
##   z      cells    name      grob
## 1 1 (1-1,1-1) arrange gtable[layout]
## 2 2 (1-1,2-2) arrange gtable[layout]
```

A superfície estimada pela nova rede parece ser uma versão ampliada ou com um pouco de zoom da figura original. Isso indica que a nova rede conseguiu capturar a tendência e a estrutura geral dos dados, reproduzindo uma superfície próximo à original.

f) Construa uma nova rede, agora ajustada sobre os valores previstos (ao invés dos valores observados de  $y$ ) para cada observação dos conjuntos de treinamento e validação. Use a arquitetura mais parcimoniosa que conseguir, sem comprometer substancialmente o poder de previsão da rede (quando comparada à obtida no item 2b). Cite um possível uso para essa nova rede.

```
set.seed(1)
indice_treinamento <- sample(1:nrow(dados), 0.6 * nrow(dados))
indice_validacao <- sample(setdiff(1:nrow(dados), indice_treinamento), 0.2 * nrow(dados))
indice_teste <- setdiff(1:nrow(dados), c(indice_treinamento, indice_validacao))

dados_treinamento <- dados[indice_treinamento, ]
dados_validacao <- dados[indice_validacao, ]
dados_teste <- dados[indice_teste, ]
```

```

x_treinamento <- as.matrix(dados_treinamento[, c("x1.obs", "x2.obs")])
y_treinamento <- as.matrix(dados_treinamento$y)

x_validacao <- as.matrix(dados_validacao[, c("x1.obs", "x2.obs")])
y_validacao <- as.matrix(dados_validacao$y)
# Calcular os valores previstos pela rede neural
y_pred_treino<-model %>% predict(as.matrix(x_treino))
y_pred_validacao<-model %>% predict(as.matrix(x_validacao))

# Criar um novo modelo
modelo_novo<-keras_model_sequential() %>%
  layer_dense(units = 64, activation = "relu", input_shape = c(2)) %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 32, activation = "relu") %>%
  layer_dropout(rate = 0.2) %>%
  layer_dense(units = 1, activation = "linear")

# Compilar o novo modelo
modelo_novo %>% compile(
  loss = "mean_squared_error",
  optimizer = optimizer_adam(),
  metrics = c("accuracy")
)

# Treinar o novo modelo com os valores previstos
historico_novo<-modelo_novo %>% fit(
  x = as.matrix(x_treino),
  y = y_pred_treino,
  epochs = 40,
  batch_size = 790,
  validation_data = list(as.matrix(x_validacao), y_pred_validacao),
  verbose = 0
)

# Calcular a precisão no conjunto de validação usando os valores previstos

precisao_novo<-modelo_novo %>% evaluate(
  x = as.matrix(x_validacao),
  y = y_pred_validacao,
  verbose = 0
)

precisao_novo

##      loss accuracy
## 2.609697 0.000000

```

Percebe-se que o modelo ajustado sobre os valores previstos apresenta uma performance semelhante ao modelo original treinado no conjunto de dados observados. Mesmo não tendo sido treinado diretamente no banco original, o modelo foi capaz de capturar as relações e padrões presentes nos dados de treinamento e gerar previsões comparáveis. Isso indica que a arquitetura da rede e os parâmetros ajustados foram eficazes

na captura das características relevantes do problema. Essa abordagem de ajustar o modelo sobre os valores previstos pode ser útil em situações em que não temos acesso direto aos valores observados da variável alvo, mas possuímos previsões de outros modelos ou estimativas provenientes de métodos diferentes. Ao ajustar um novo modelo sobre essas previsões, podemos aproveitar as informações disponíveis e obter uma estimativa mais refinada da variável alvo. Isso pode ser particularmente útil em casos em que os modelos anteriores já foram validados e demonstraram bom desempenho em problemas relacionados. Essa rede pode ser útil na construção de ensembles, que é uma técnica de combinação de modelos. Nesse contexto, podemos utilizar as previsões de diferentes modelos como entrada para uma nova rede neural, que realiza a combinação ou refinamento dessas previsões. Ao utilizar as previsões de diferentes modelos como entrada para a nova rede, estamos aproveitando a diversidade de perspectivas e abordagens dos modelos individuais.