Jared Yu
Final

Problem 4

a. Show a neural network that calculates p16.

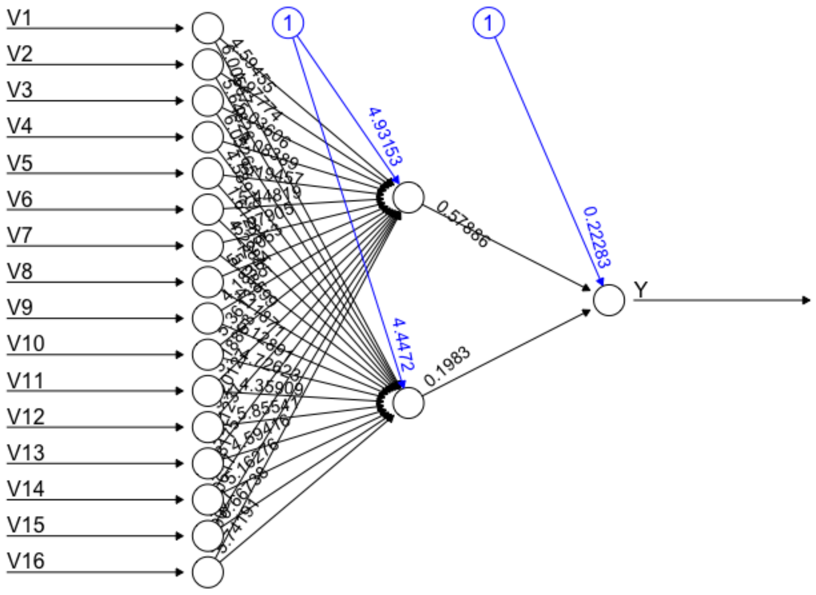
Ans:

For this problem, the first step was to create the truth table in R. Afterwards, these truth table and the corresponding parity_16 result could be plugged into the neural network from the previous homework as the X and Y. The neural network here utilizes the "neuralnet" package in R. (Note: the following uses code from @158 in Piazza)

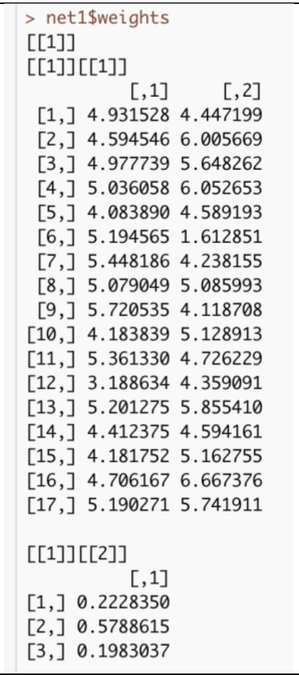
The code used is seen below:

```
X <- empty_mat_df

Y <- a_or_b_or_not_a_and_b

df <- cbind(Y, X)

net1 <- neuralnet(Y~., df, hidden=2)

plot(net1)
```

Here, X are the $x_1, \cdots, x_{16}$ values of the truth table, and Y is p_{16}. It creates a neural network with 2 layers. The following is a visualization of the neural network:



Below are the weights in the image:

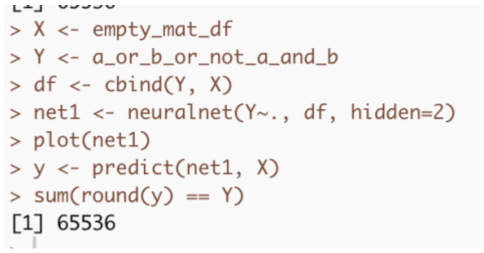


```
> net1$weights
[[1]]
[[1]][[1]]
          [,1]     [,2]
 [1,] 4.931528 4.447199
 [2,] 4.594546 6.005669
 [3,] 4.977739 5.648262
 [4,] 5.036058 6.052653
 [5,] 4.083890 4.589193
 [6,] 5.194565 1.612851
 [7,] 5.448186 4.238155
 [8,] 5.079049 5.085993
 [9,] 5.720535 4.118708
[10,] 4.183839 5.128913
[11,] 5.361330 4.726229
[12,] 3.188634 4.359091
[13,] 5.201275 5.855410
[14,] 4.412375 4.594161
[15,] 4.181752 5.162755
[16,] 4.706167 6.667376
[17,] 5.190271 5.741911

[[1]][[2]]
          [,1]
[1,] 0.2228350
[2,] 0.5788615
[3,] 0.1983037
```

To double-check that everything works, the predict() function is used on the neural network and the training data:

```
y <- predict(net1, X)

round(y) == Y
```

The following shows that the code works and all the training input results in the correct output:



```
> X <- empty_mat_df
> Y <- a_or_b_or_not_a_and_b
> df <- cbind(Y, X)
> net1 <- neuralnet(Y~., df, hidden=2)
> plot(net1)
> y <- predict(net1, X)
> sum(round(y) == Y)
[1] 65536
```

It indicates that all 65,536 rows of the truth table evaluated correctly.

b. Show a neural network that calculates p_{127}.

The professor mentioned that it's not possible to train directly a network for p_{128}, since it's simply too large to save in memory for a normal computer. However, some special architecture can be used to take advantage of the symmetry of the problem.

c. Write down the gradient descent update equations for a network to calculate p_8.

The following is directly copied from a previous module 7 homework assignment:

The first step is to show that $\nabla \mathbf{J}_q(\mathbf{a}_k) = 2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1$. In the case where $\mathcal{Y}(\mathbf{a}_k)$ only contains $\mathbf{y}_1$, then $J_q(\mathbf{a}) = (\mathbf{a}^\top \mathbf{y}_1 - b)^2$. Finding the derivative of this w.r.t. $\mathbf{a}$, we find that,

$$\frac{\partial}{\partial \mathbf{a}_k} J_q(\mathbf{a}_k) = \frac{\partial}{\partial \mathbf{a}_k}(\mathbf{a}_k^\top \mathbf{y}_1 - b)^2 = 2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\frac{\partial}{\partial \mathbf{a}_k}(\mathbf{a}_k^\top \mathbf{y}_1 - b) = 2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1.$$

To find the matrix of second partial derivatives, we can take the partial derivative again to see that,

$$\frac{\partial^2}{\partial \mathbf{a}_k^\top \partial \mathbf{a}_k} J_q(\mathbf{a}_k) = \frac{\partial}{\partial \mathbf{a}_k^\top} 2\mathbf{y}_1(\mathbf{a}_k^\top \mathbf{y}_1 - b) = 2\frac{\partial}{\partial \mathbf{a}_k^\top}(\mathbf{y}_1 \mathbf{a}_k^\top \mathbf{y}_1 - b\mathbf{y}_1)$$
$$= 2\mathbf{y}_1 \frac{\partial}{\partial \mathbf{a}_k^\top}(\mathbf{a}_k^\top \mathbf{y}_1) = 2\mathbf{y}_1\mathbf{y}_1^\top = D$$

To find $\mathbf{a}_{k+1} = \mathbf{a}_k - \rho_k \nabla \mathbf{J}(\mathbf{a}_k)$, we can use the formula for ρ_k from the textbook.

$$\rho_k = \frac{\left\|\nabla \mathbf{J}_q(\mathbf{a}_k)\right\|^2}{\nabla \mathbf{J}_q(\mathbf{a}_k)^\top D \nabla \mathbf{J}_q(\mathbf{a}_k)} = \frac{\|2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1\|^2}{[2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1]^\top [2\mathbf{y}_1\mathbf{y}_1^\top][2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1]}$$
$$= \frac{4(\mathbf{a}_k^\top \mathbf{y}_1 - b)^2 \mathbf{y}_1^\top \mathbf{y}_1}{8(\mathbf{a}_k^\top \mathbf{y}_1 - b)^2 \mathbf{y}_1^\top \mathbf{y}_1 \mathbf{y}_1^\top \mathbf{y}_1} = \frac{1}{2\mathbf{y}_1^\top \mathbf{y}_1} = \frac{1}{2\|\mathbf{y}_1\|^2}$$

Then, going back to the update formula we have the following,

$$\mathbf{a}_{k+1} = \mathbf{a}_k - \rho_k \nabla \mathbf{J}(\mathbf{a}_k)$$
$$= \mathbf{a}_k - \frac{\nabla \mathbf{J}(\mathbf{a}_k)}{2\|\mathbf{y}_1\|^2}$$
$$= \mathbf{a}_k - \frac{2(\mathbf{a}_k^\top \mathbf{y}_1 - b)\mathbf{y}_1}{2\|\mathbf{y}_1\|^2}$$
$$= \mathbf{a}_k + \frac{b - \mathbf{a}^\top \mathbf{y}_1}{\|\mathbf{y}_1\|^2}\mathbf{y}_1 \blacksquare$$

It shows the update equation for gradient descent, but it doesn't specify the exact error function for this test problem.

d. Train such a neural network and show weights and the output for each of

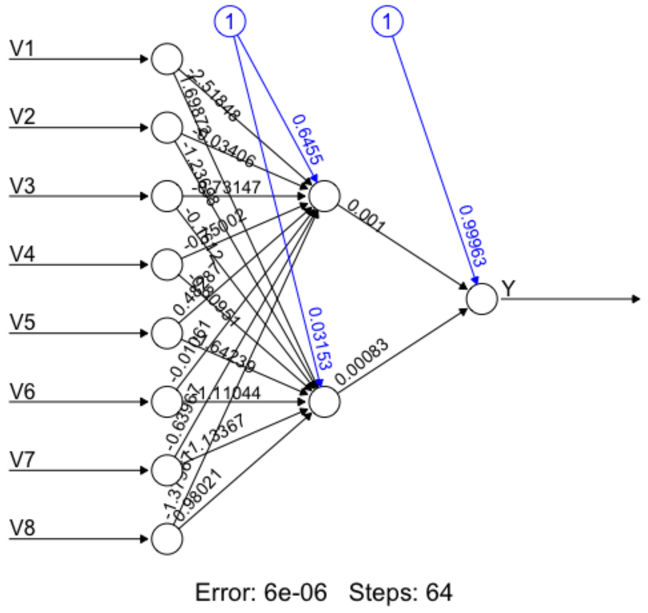
Ans:

Here, the steps were to repeat what was done in part a with p_{16}, however a truth table was first generated instead for p_8. Then the code below as used in R:

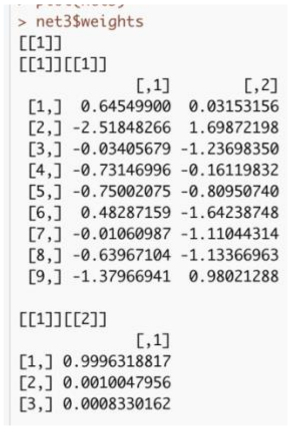
```
X <- empty_mat_df

Y <- a_or_b_or_not_a_and_b

df <- cbind(Y, X)

net3 <- neuralnet(Y~., df, hidden=2)

plot(net3)
```

```
y <- predict(net3, X)

sum(round(y) == Y)
```

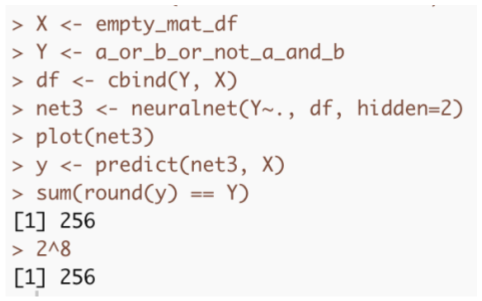
This leads to the following network image:



The weights for the above network can be seen below:

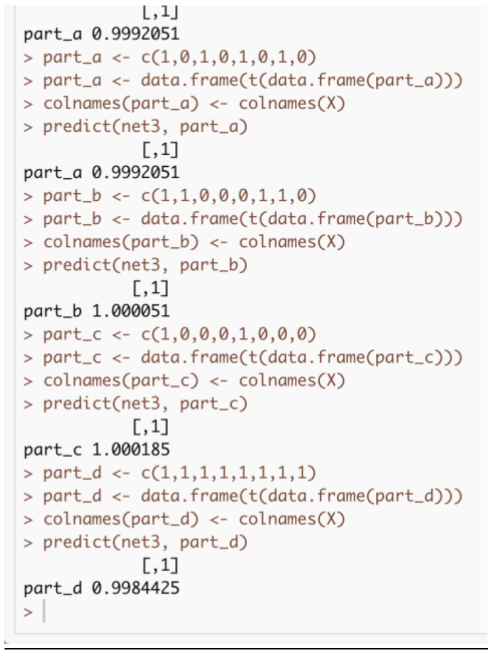

```
> net3$weights
[[1]]
[[1]][[1]]
            [,1]        [,2]
 [1,]  0.64549900  0.03153156
 [2,] -2.51848266  1.69872198
 [3,] -0.03405679 -1.23698350
 [4,] -0.73146996 -0.16119832
 [5,] -0.75002075 -0.80950740
 [6,]  0.48287159 -1.64238748
 [7,] -0.01060987 -1.11044314
 [8,] -0.63967104 -1.13366963
 [9,] -1.37966941  0.98021288

[[1]][[2]]
             [,1]
[1,] 0.9996318817
[2,] 0.0010047956
[3,] 0.0008330162
```

The code below shows that it all works correctly and the training observations are all mapped without error:



```
> X <- empty_mat_df
> Y <- a_or_b_or_not_a_and_b
> df <- cbind(Y, X)
> net3 <- neuralnet(Y~., df, hidden=2)
> plot(net3)
> y <- predict(net3, X)
> sum(round(y) == Y)
[1] 256
> 2^8
[1] 256
```

In the following parts a-d, the output is just showing what happens when the following inputs are sent through the neural network and what their resulting probabilities are. The following screenshot shows the output for when inserting each of the vectors from parts a-d into the neural network:



```
            [,1]
part_a 0.9992051
> part_a <- c(1,0,1,0,1,0,1,0)
> part_a <- data.frame(t(data.frame(part_a)))
> colnames(part_a) <- colnames(X)
> predict(net3, part_a)
            [,1]
part_a 0.9992051
> part_b <- c(1,1,0,0,0,1,1,0)
> part_b <- data.frame(t(data.frame(part_b)))
> colnames(part_b) <- colnames(X)
> predict(net3, part_b)
           [,1]
part_b 1.000051
> part_c <- c(1,0,0,0,1,0,0,0)
> part_c <- data.frame(t(data.frame(part_c)))
> colnames(part_c) <- colnames(X)
> predict(net3, part_c)
           [,1]
part_c 1.000185
> part_d <- c(1,1,1,1,1,1,1,1)
> part_d <- data.frame(t(data.frame(part_d)))
> colnames(part_d) <- colnames(X)
> predict(net3, part_d)
            [,1]
part_d 0.9984425
>
```

a. 10101010,

In the above screenshot for part a, the output is 0.9992051.

b. 11000110,

In the above screenshot for part b, the output is 1.000051.

c. 10001000,

In the above screenshot for part c, the output is 1.000185.

d. 11111111,

In the above screenshot for part d, the output is 0.9984425.

Problem 5

Here, the goal is to generate the mixture of normal variables. This was done by first generating X_1, X_2, and X_3 along with their prior probabilities. This was done using the following in R:

```
N <- 500

X1 <- rnorm(n = N, mean = 1, sd = sqrt(0.1))

X2 <- rnorm(n = N, mean = 2, sd = sqrt(0.1))

X3 <- rnorm(n = N, mean = 3, sd = sqrt(0.2))

P1 <- 1/6

P2 <- 1/2

P3 <- 1/3
```

Afterwards the mixture can be generated using the following code:

X_mix <- (X1 * P1) + (X2 * P2) + (X3 * P3)

Then, using the EM package “mclust,” which was also used in the previous homework, the following code was used:

em_clustering = Mclust(data = X_mix, G = 3)

em_clustering$parameters

This fits the Mclust function to the data, with the number of groups set to 3.

For the mean, variance, and priors, the estimates are as follows:

j	μ_j	σ_j^2	P_j
1	2.001067	0.03335327	0.359207692
2	2.249608	0.03335327	0.634304789
3	2.799298	0.03335327	0.006487518

It is interesting to see that the package only allows for a default prior being equal for all clusters unless something is specified. Since it is being assumed that they are unknown, then the package returns the default value for all of them. It seems that in the univariate case that the algorithm has trouble trying to identify correctly the unknown parameters, as there are some noticeable differences from their true values.

Code Appendix

```
### Problem 4
### parity (16)
parity_n=16
empty_mat <- matrix(NA, nrow = 2^parity_n, ncol = parity_n)

# count down inside count up outside
inner_count <- parity_n; outer_count <- 0
for (i in 1:parity_n) {
  # print(paste0('i', inner_count - i, 'o', outer_count + i))
  new_vec <- rep(rep(c(1, 0), each = 2^(inner_count - i)), 2^(outer_count + i - 1))
  empty_mat[,i] <- new_vec
}

empty_mat_df <- as.data.frame(empty_mat)

# XOR
# (A AND B)
a_and_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 AND 2
    a_and_b <- empty_mat_df[,i] & empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 AND 3,
    a_and_b <- empty_mat_df[,i] & a_and_b
  }
}
a_and_b <- a_and_b * 1

# (A OR B)
a_or_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 OR 2
    a_or_b <- empty_mat_df[,i] | empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 OR 3,
    a_or_b <- empty_mat_df[,i] | a_or_b
  }
}
a_or_b <- a_or_b * 1

not_a_and_b <- (!a_and_b) * 1

# (A OR B) OR NOT (A AND B)
a_or_b_or_not_a_and_b <- (a_or_b | not_a_and_b) * 1

library(neuralnet)
X <- empty_mat_df
Y <- a_or_b_or_not_a_and_b
df <- cbind(Y, X)
net1 <- neuralnet(Y~., df, hidden=2)
plot(net1)

y <- predict(net1, X)
sum(round(y) == Y)

### parity(127)
parity_n <- 127
empty_mat <- matrix(0, nrow = 2^parity_n, ncol = parity_n)

# count down inside count up outside
inner_count <- parity_n; outer_count <- 0
```

```r
for (i in 1:parity_n) {
  # print(paste0('i', inner_count - i, 'o', outer_count + i))
  new_vec <- rep(rep(c(1, 0), each = 2^(inner_count - i)), 2^(outer_count + i - 1))
  empty_mat[,i] <- new_vec
}

empty_mat_df <- as.data.frame(empty_mat)
write.csv(empty_mat_df, file = 'truth_table127.csv', row.names = FALSE)

# XOR
# (A AND B)
a_and_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 AND 2
    a_and_b <- empty_mat_df[,i] & empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 AND 3,
    a_and_b <- empty_mat_df[,i] & a_and_b
  }
}
a_and_b <- a_and_b * 1

# (A OR B)
a_or_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 OR 2
    a_or_b <- empty_mat_df[,i] | empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 OR 3,
    a_or_b <- empty_mat_df[,i] | a_or_b
  }
}
a_or_b <- a_or_b * 1

not_a_and_b <- (!a_and_b) * 1

# (A OR B) OR NOT (A AND B)
a_or_b_or_not_a_and_b <- (a_or_b | not_a_and_b) * 1

### parity (8)
parity_n=8
empty_mat <- matrix(NA, nrow = 2^parity_n, ncol = parity_n)

# count down inside count up outside
inner_count <- parity_n; outer_count <- 0
for (i in 1:parity_n) {
  # print(paste0('i', inner_count - i, 'o', outer_count + i))
  new_vec <- rep(rep(c(1, 0), each = 2^(inner_count - i)), 2^(outer_count + i - 1))
  empty_mat[,i] <- new_vec
}

empty_mat_df <- as.data.frame(empty_mat)

# XOR
# (A AND B)
a_and_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 AND 2
    a_and_b <- empty_mat_df[,i] & empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 AND 3,
    a_and_b <- empty_mat_df[,i] & a_and_b
  }
}
a_and_b <- a_and_b * 1
```

```r
# (A OR B)
a_or_b <- rep(NA, 2^parity_n)
for (i in 1:(parity_n - 1)) {
  if (i == 1) {
    # 1 OR 2
    a_or_b <- empty_mat_df[,i] | empty_mat_df[,i+1]
  } else if ((i != 1) & (i != 2)) {
    # 2 OR 3,
    a_or_b <- empty_mat_df[,i] | a_or_b
  }
}
a_or_b <- a_or_b * 1

not_a_and_b <- (!a_and_b) * 1

# (A OR B) OR NOT (A AND B)
a_or_b_or_not_a_and_b <- (a_or_b | not_a_and_b) * 1

X <- empty_mat_df
Y <- a_or_b_or_not_a_and_b
df <- cbind(Y, X)
net3 <- neuralnet(Y~., df, hidden=2)
plot(net3)
y <- predict(net3, X)
sum(round(y) == Y)
2^8

part_a <- c(1,0,1,0,1,0,1,0)
part_a <- data.frame(t(data.frame(part_a)))
colnames(part_a) <- colnames(X)
predict(net3, part_a)

part_b <- c(1,1,0,0,0,1,1,0)
part_b <- data.frame(t(data.frame(part_b)))
colnames(part_b) <- colnames(X)
predict(net3, part_b)

part_c <- c(1,0,0,0,1,0,0,0)
part_c <- data.frame(t(data.frame(part_c)))
colnames(part_c) <- colnames(X)
predict(net3, part_c)

part_d <- c(1,1,1,1,1,1,1,1)
part_d <- data.frame(t(data.frame(part_d)))
colnames(part_d) <- colnames(X)
predict(net3, part_d)

### Problem 5
library(mclust)
N <- 500
X1 <- rnorm(n = N, mean = 1, sd = sqrt(0.1))
X2 <- rnorm(n = N, mean = 2, sd = sqrt(0.1))
X3 <- rnorm(n = N, mean = 3, sd = sqrt(0.2))
P1 <- 1/6
P2 <- 1/2
P3 <- 1/3

X_mix <- (X1 * P1) + (X2 * P2) + (X3 * P3)
em_clustering = Mclust(data = X_mix, G = 3)
em_clustering$parameters
```