Conceptual/theoretical problems

1. Prove the last two equations governing backpropagation., BP3 and BP4.

   We can define the cost function as:

$$\frac{1}{2N}\sum_{i=1}^{N}||\vec{y}_i - \vec{a}^L(\vec{x}_i)||^2$$

   We can also define $\vec{a}^L$ in the following way. This will help us conceptualize the chain rule.

$$\vec{a}^L = \sigma\Big(W^L\,\sigma\Big(W^{L-1}\,\sigma\big(\dots\sigma(W^1a_0+b^1)\dots\big)+b^{L-1}\Big)+b^L\Big)$$

   where σ is the sigmoid-logistic function, W^j is the j^{th} weight matrix, and b^j is the j^{th} bias vector. Let's begin by proving BP3.

   Proof. We are trying to find the partial derivative of the cost function with respect to the j^{th} bias vector. Using the same notation from lecture, we can show the following...

$$\frac{\partial C}{\partial b_j^\ell} = \sum_{j\,\in\,\text{batch}} \frac{\partial C}{\partial a_j^\ell}\cdot\frac{\partial a_j^\ell}{\partial z_j^\ell}\cdot\frac{\partial z_j^\ell}{\partial b_j^\ell}$$

   However, the partial derivative of z_j^ℓ with respect to each beta vector is 1 if $i \in$ batch and 0 otherwise. Hence, we just end up with

$$\frac{\partial C}{\partial b_j^\ell} = \sum_{j\,\in\,\text{batch}} \frac{\partial C}{\partial a_j^\ell}\cdot\frac{\partial a_j^\ell}{\partial z_j^\ell}$$

   and if we follow the same steps as BP2 we can arrive at the following:

$$\begin{aligned}
\sum_{j\,\in\,\text{batch}} \frac{\partial C}{\partial a_j^\ell}\cdot\frac{\partial a_j^\ell}{\partial z_j^\ell} &= \frac{\partial C}{\partial a_j^\ell}\cdot\frac{\partial a_j^\ell}{\partial z_j^\ell} \\
&= \left(\sum_{i\,\in\,\text{batch}} \frac{\partial C}{\partial z_i^{\ell+1}}\cdot\frac{\partial z_i^{\ell+1}}{\partial a_j^\ell}\right)\cdot\sigma'\left(z_j^\ell\right) \\
&= \left(\sum_{i\,\in\,\text{batch}} \delta_i^{\ell+1}W_{ji}^{\ell+1}\right)\cdot\sigma'\left(z_j^\ell\right) \\
&= \vec{\delta}^{\ell+1}{W^{\ell+1}}^T\odot\sigma'\left(\vec{z}^\ell\right) \\
&= \vec{\delta}^\ell
\end{aligned}$$

   Thus,

$$\frac{\partial C}{\partial \vec{b}^\ell} = \vec{\delta}^\ell$$

   □

We can use the same process for proving BP4.

Proof. We are trying to find the partial derivative of the cost function with respect to the j^{th} weight matrix. Using the same notation from lecture, we can show the following...

$$\frac{\partial C}{\partial W_{jk}^{\ell}} = \sum_{j \in \text{batch}} \frac{\partial C}{\partial a_j^{\ell}} \cdot \frac{\partial a_j^{\ell}}{\partial z_j^{\ell}} \cdot \frac{\partial z_j^{\ell}}{\partial W_{jk}^{\ell}}$$

The partial derivative of z_j^{ℓ} with respect to each weight matrix is $a_k^{\ell-1}$ if $i \in$ batch and 0 otherwise. Hence, we just end up with

$$\frac{\partial C}{\partial W_{jk}^{\ell}} = a_k^{\ell-1} \sum_{j \in \text{batch}} \frac{\partial C}{\partial a_j^{\ell}} \cdot \frac{\partial a_j^{\ell}}{\partial z_j^{\ell}}$$

and if we follow the same steps as BP2 we can arrive at the following:

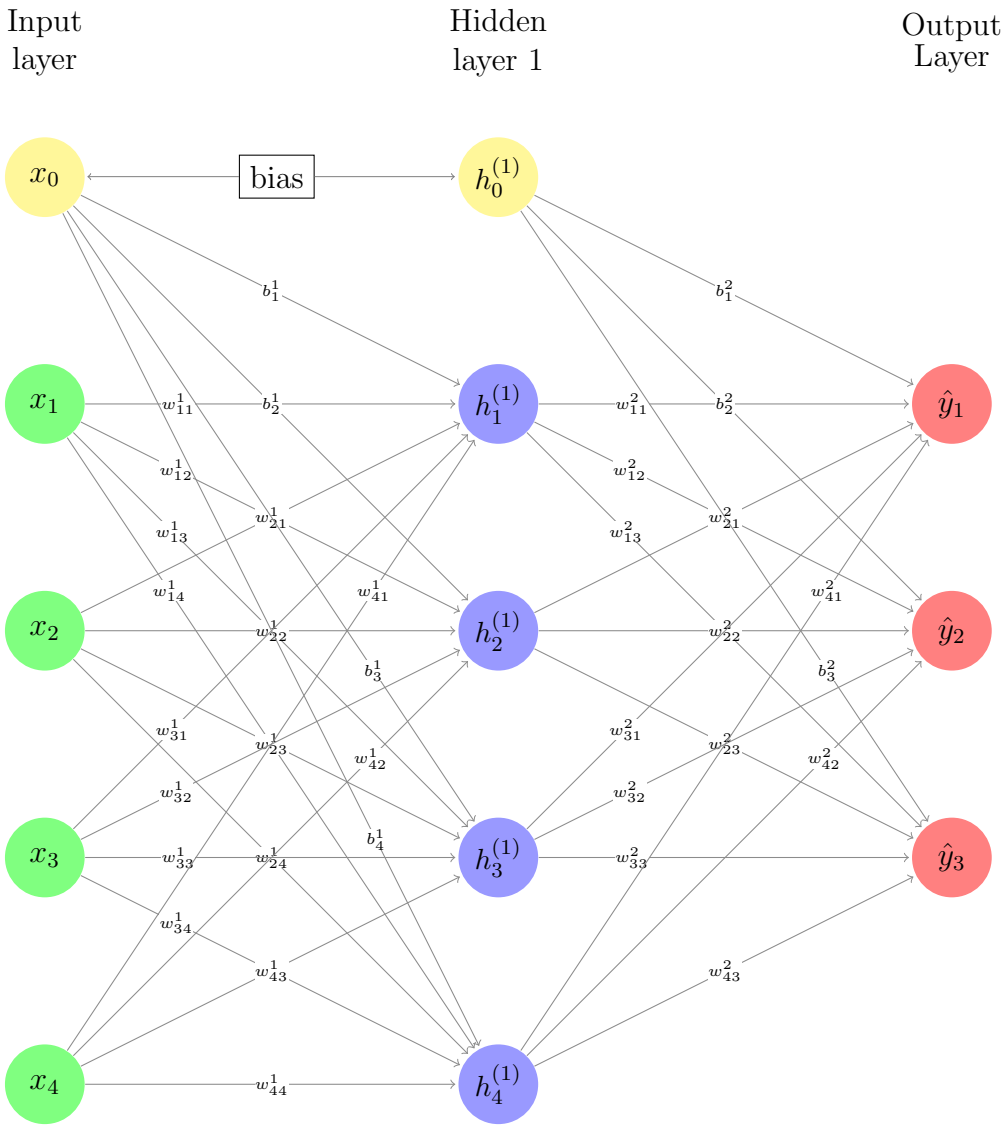
$$\begin{aligned}
a_k^{\ell-1} \sum_{j \in \text{batch}} \frac{\partial C}{\partial a_j^{\ell}} \cdot \frac{\partial a_j^{\ell}}{\partial z_j^{\ell}} &= a_k^{\ell-1} \frac{\partial C}{\partial a_j^{\ell}} \cdot \frac{\partial a_j^{\ell}}{\partial z_j^{\ell}} \\
&= a_k^{\ell-1} \left(\sum_{i \in \text{batch}} \frac{\partial C}{\partial z_i^{\ell+1}} \cdot \frac{\partial z_i^{\ell+1}}{\partial a_j^{\ell}} \right) \cdot \sigma' \left(z_j^{\ell} \right) \\
&= a_k^{\ell-1} \left(\sum_{i \in \text{batch}} \delta_i^{\ell+1} W_{ji}^{\ell+1} \right) \cdot \sigma' \left(z_j^{\ell} \right) \\
&= a_k^{\ell-1} \left(\vec{\delta}^{\ell+1} W^{\ell+1^T} \odot \sigma' \left(\vec{z}^{\ell} \right) \right) \\
&= \vec{a}^{\ell-1^T} \vec{\delta}^{\ell}
\end{aligned}$$

Thus,

$$\frac{\partial C}{\partial W^{\ell}} = \vec{a}^{\ell-1^T} \vec{\delta}^{\ell}$$

□

8. The backpropagation algorithm is difficult to code (though I think you are capable), so for this portion, we'll try something different. My function is on the next page. Explain it back to me, line-by-line, including the helper function. Write a detailed explanation of the purpose of each line, connecting it to the mathematics from lecture wherever possible. I'm expecting more depth than the outline of Algorithm 9.3.



Above is a graph of the neural network that we train in Example 9.4. The goal of this algorithm is to train data to find the best weights and biases that make predictions for input values.

```
backPropagate <- function(X, G, hidden, rate, iterations, batchSize){
```

These are the inputs for the function. We input the data matrix `X` consisting of the data we want to make a prediction for. This is the "training" data. The input `G` is a binary response

matrix, or a categorical vector. It will be converted into a binary response matrix within the function. The input `hidden` will be the number of hidden layers and number of nodes in each. For instance, if you want three hidden layers with 4 nodes, 5 nodes, and 4 nodes you would input: `c(4,5,4)`. The input `rate` is the step we want to make in the gradient descent step. Too big of a rate and the function will not converge, and too little of a rate and it may also never converge. The input `iterations` will be the number of loops we want to run; the more iterations, the better the network (may take a long time to build though). Lastly, the input `batchsize` will be the number of points we want to sample for each iteration out of the total sample size.

```
G <- as.matrix(G)
bestCost <- Inf
X <- scale(X)
Y <- transformGtoY(G)
neuralNetwork <- initializeNetwork(ncol(X), hidden, unique(G))
L <- length(hidden) + 1
```

For these couple lines we are essentially preparing the function to run through the iterations. We are turning `G` into a matrix to help the later function `transformGtoY`. Since we are wanting to minimize the cost function, we start with cost of ∞ just to initialize it. We do not want to pick any arbitrary large number, say a billion, because a big enough network may have a cost function over a billion, which in that case may never converge. Next, we are scaling `X` because the backpropagation will perform better with standardized data. Next, we are turning `G` into a binary response matrix and calling it `Y`. The next line is just initializing an "empty" neural network to fill in for each iteration. It is creating weight matrices and bias vectors with the correct dimensions and entries of random normally distributed data close to 0. We then define `L` as the total number of layers, which includes every hidden layer and the one output layer.

```
for (i in 1:iterations) {
```

Now we are going to loop through each iteration. The more iterations, the more efficient the neural network should work. The goal is to minimize the cost function, while also not being too expensive for the computer. Too many iterations will be expensive. I settled on 10,000 at first, but you advised 50,000. This gives a well-trained network.

```
    indices <- sample(nrow(X), batchsize)
    tempX <- X[indices,]
    tempY <- Y[indices,]
```

These lines will sample data from the input data and shrink it to our choice of `batchsize`. If we were to pick a batchsize of 100, it will randomly choose 100 numbers between one and the number of rows in the data matrix. Then it will use tht numbers (`indices`) as the indices of the rows of `X` and `Y`.

```
    AandZ <- feedForward(tempX, neuralNetwork, backPropagate = TRUE)
```

This line is finding the `Alist` and `Zlist` using the `feedForward` function. The inputs for the function will use the slimmed-down data matrix from the previous few lines, it will use the neural network from the previous loop (for the first loop it will use the neural network we initialized earlier), and we set `backPropagate = TRUE`, which will output the `Alist` and

`Zlist` matrices. Recall that the `Alist` matrices will be the activation outputs from each layer and the `Zlist` matrices will be the inputs to the sigmoid function.

```
delta <- (AandZ$Alist[[L]]-tempY)*sigmoidPrime(AandZ$Zlist[[L]])
Bgrad <- colMeans(delta)
Wgrad <- WgradHelper(AandZ$Alist[[L-1]], delta)
```

The first line is implementing the formula BP1 from lecture. The following line is then finding the column means for each, which will give the average `delta` for each iteration. The last line of this bit is implementing BP4 from lecture and finding the gradient for the weight. This line is using the formula `WgradHelper`, which we will go over later. It is calculating an "outer" product between the activation of the `L-1` layer and the delta vector.

```
neuralNetwork$Blist[[L]] <- neuralNetwork$Blist[[L]] - rate*Bgrad
neuralNetwork$Wlist[[L]] <- neuralNetwork$Wlist[[L]] - rate*Wgrad
```

These two lines are using the formula from stochastic gradient descent to update the weight matrix and bias vector for the last layer. After this last layer has been updated, we move into looping through each hidden layer via BP2.

```
for (l in (L-1):1) {
```

Now we will begin to loop through the hidden layers, starting with the second to last layer (`L - 1`) and going to the first hidden layer.

```
  delta <- delta %*%
    t(neuralNetwork$Wlist[[l+1]])*sigmoidPrime(AandZ$Zlist[[l]])
  Bgrad <- colMeans(delta)
```

These lines are finding the gradient of the bias vector using BP2 of the ℓ^{th} layer, similar to before.

```
  if (l == 1) {
    Wgrad <- WgradHelper(tempX, delta)
  }
  else {
    Wgrad <- WgradHelper(AandZ$Alist[[l-1]], delta)
  }
```

Notice that when we get to the last layer when calculating the gradient of the weights, we are using `AandZ$Alist[[l-1]]`, which in this case doesn't exists. That's why we need to set up an `if` statement for the last loop (`l == 1`). When we are at the last loop we use a_0, or in this case `tempX`.

```
  neuralNetwork$Blist[[l]] <- neuralNetwork$Blist[[l]] - rate*Bgrad
  neuralNetwork$Wlist[[l]] <- neuralNetwork$Wlist[[l]] - rate*Wgrad
```

In these two lines we are updating the neural network for each iteration of the ℓ^{th} layer, similar to before

```
a <- X
```

This is initializing the matrix `a` with the same dimensions as `X`.

```
  for (l in 1:L) {
    a <- sigmoid(sweep(a %*% neuralNetwork$Wlist[[l]], 2,
        neuralNetwork$Blist[[l]], "+"))
  }
```

This `for` loop is "sweep"ing over every layer and calculating the activation. It's using the formula $\vec{a}^{\ell} = \sigma\left(\vec{a}^{\ell-1} \cdot W^{\ell} + \vec{b}^{\ell}\right)$. We need this value when trying to find the value of the cost function.

```
  cost <- sum((a - Y)^2)
  print(cost)

  if (cost < bestCost) {
    bestNetwork <- neuralNetwork
    bestCost <- cost
  }
```

In these few lines we are simply calculating the cost function, printing it, and then updating the best network and best cost if the current cost value is less than the previous loop. This way the best network and best cost will always be the best it can. At this point all of the dirty work is over.

```
  return(bestNetwork)
```

Self-explanatory.

Now we will go over the function `WgradHelper`, which stands for Weight Gradient Helper. This function will be calculating BP4 for each iteration.

```
  WgradHelper <- function(a, d){
```

This function inputs two matrices. In our context the first input is the `Alist` matrix from the $\ell - 1$ layer. The next input will be the delta matrix from the ℓ^{th} layer.

```
  sum <- matrix(0, nrow = ncol(a), ncol = ncol(d))
  for (i in 1:nrow(a)) {
    sum <- sum + outer(a[i,], d[i,])
  }
  sum <- sum/nrow(a)
```

Recall that the formula for BP4 is $\frac{\partial C}{\partial W^{\ell}} = \vec{a}^{\ell-1^t} \cdot \vec{\delta}^{\ell}$. We are multiplying a column vector to a row vector. We call this an outer product, and we will end up with a matrix with the number of rows corresponding to the number of rows in the column vector, and number of columns corresponding to the number of entries in the row vector. This function first initializes a matrix of this size and calls it `sum`. It then loops through the number of rows of the `a` matrix. It will calculate the outer product of each row of the `Alist` matrix and each row of the `delta` matrix. Each loop it will add this to `sum`. Each loop the `sum` will grow. After this calculation is done for each row, we are then left with a matrix that has been added through `nrow(a)` times. By dividing by this same number, we will be left with the average. This calculation comes from the stochastic gradient descent calculation, $\hat{\nabla} f = \frac{\sum_{i \in \text{batchsize}} \nabla f_i}{\text{batchsize}}$.