Assignment 1

1 Softmax

1. **Show that softmax(x) = softmax(x + c).**

This exercise is not merely academic. This property allows us to substract the maximum element in a vector from all other elements in that vector, and then apply softmax to the result, without affecting the softmax values. This is important to ensure numerical stability. Without normalization, the elements e^{x_i} can be very large values that lead to numerical overflow. With normalization, all elements are mapped to values between 0 and 1.

$$\begin{aligned} softmax(x+c)_j &= \frac{e^{x_j+c}}{\sum_i e^{x_i+c}} \\ &= \frac{e^{x_j}e^{c}}{\sum_i e^{x_i}e^{c}} \\ &= \frac{e^{x_j}}{\sum_i e^{x_i}} \\ &= sofmax(x)_j \end{aligned}$$

2. **Implement the softmax function in q1_softmax.py.**

We first define a helper function, `softmax_vector(v)`, that calculates the element-wise softmax values of the vector `v`. This function uses the property we proved above.

```
def softmax_vector(v):
  m = np.max(v)
  return np.exp(v - m)/np.sum(np.exp(v - m))
```

There are two cases: when `x` is a matrix and when `x` is a vector. If `x` is a matrix, we apply `softmax_vector` row-wise, using the function `np.apply_along_axis`. If `x` is a vector, we just apply `softmax_vector` directly.

```
if len(x.shape) > 1:
  # Matrix
  ### YOUR CODE HERE
  x = np.apply_along_axis(softmax_vector, 1, x)
  ### END YOUR CODE
else:
  # Vector
  ### YOUR CODE HERE
  x = softmax_vector(x)
  ### END YOUR CODE
```

2 Neural Network Basics

1. **Derive the gradients of the sigmoid function and show that it can be rewritten as a function of the function value**

The only "trick" here is the addition and subtraction of e^{-x} in line 3.

$$\begin{aligned}
\sigma\prime(x) &= \frac{e^{-x}}{(1+e^{-x})^2} \\
&= \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1}{1+e^{-x}} \\
&= \frac{e^{-x}}{1+e^{-x}} \cdot \frac{1+e^{-x}-e^{-x}}{1+e^{-x}} \\
&= \sigma(x)(1-\sigma(x))
\end{aligned}$$

2. **Derive the gradient with regard to the inputs of a softmax function when cross entropy loss is used for evaluation**

This gradient is the first step in the back-propagation of our neural network. Every other gradient we calculate during backprop will contain this gradient as a factor. As the hint points out, y is a one-hot vector, so the only term j that contributes to the gradient is where $y_j = 1$. All other terms are zero. This simplifies the expressions to

$$\begin{aligned}
CE(y, \hat{y}) &= -\sum_i y_i \log(\hat{y}_i) \\
&= \log(softmax(\theta)_j) \\
&= \log(\frac{e^{\theta_j}}{\sum_k e^{\theta_k}}) \\
&= -\theta_j + \log(\sum_k e^{\theta_k})
\end{aligned}$$

The gradient with respect to θ_j is then

$$\begin{aligned}
\frac{\partial CE(y, \hat{y})}{\partial \theta_j} &= -1 + \frac{1}{\sum_k e^{\theta_k}} \cdot e^{\theta_j} \\
&= -1 + softmax(\theta)_j \\
&= \hat{y}_j - y_j
\end{aligned}$$

For $k \neq j$, the only difference is that the term -1 is instead 0, so the gradient is just $softmax(\theta)_k$. Putting all gradients together in vector notation, we get $\frac{\partial CE(y,\hat{y})}{\partial \theta} = \hat{y} - y$.

3. **Derive the gradients with respect to the inputs x to an one-hidden-layer neural network**

Book keeping is really important here, both conceptually and computationly. Conceptually, it will help to break out the applications of the chain rule into manageable chunks. Computationly, it shows what computations can be stored and re-used later. So we'll start by

writing down all the intermediate variables we'll need. Going from the output layer to the input layer, we have:

$$\begin{aligned}
\hat{y} &= softmax(\theta) \\
\theta &= hW_2 + b_2 \\
h &= \sigma(\phi) \\
\phi &= xW_1 + b_1
\end{aligned}$$

Picking up where we left off in part 1, we get

$$\begin{aligned}
\frac{\partial J}{\partial \theta} &= \hat{y} - y \\
\frac{\partial J}{\partial h} &= \frac{\partial J}{\partial \theta}\frac{\partial \theta}{\partial h} = (\hat{y} - y)W_2^T \\
\frac{\partial J}{\partial \phi} &= \frac{\partial J}{\partial h}\frac{\partial h}{\partial \phi} = (\hat{y} - y)W_2^T \circ \sigma'(\phi) \\
\frac{\partial J}{\partial x} &= \frac{\partial J}{\partial \phi}\frac{\partial \phi}{\partial x} = (\hat{y} - y)W_2^T \circ \sigma'(\phi)W_1^T
\end{aligned}$$

Notice how some values computed during forward propagation, such as ϕ and $\hat{y}$, are re-used during backprop. Also note that the derivative of a function that is applied element-wise during forward propagation is multiplied element-wise during backprop, as indicated by the Hadamard product ($\circ$).

Let's take a moment to appreciate the oddness of this question. After all, in the backprop algorithm we don't calculate the gradient w.r.t. the input data, x, which we consider to be fixed. Rather, the gradient is calculated w.r.t. the *parameters* W_1, b_1, W_2, b_2. Put another way, it's the parameter values that we adjust, via backprop and gradient descent, in order to minimize the loss function. Indeed, when we implement backprop below, we do so in terms of the parameter gradients. The work we've done in this problem is not used. Sad!