1. Question 1

A neural network basically consists of an input and output layer, with some hidden layers in between. Neural Networks can approximate and represent any function given a large enough layer, and desired error margin.

1.1. Forward-Propagation

In forward-propagation, we use the values of the previous layer in order to obtain the values of the present layer. The value of each unit is evaluated by taking a weighted sum of all units on the previous layer, and then passing an activation function on the sum.
For example, to find the value of a unit on the second layer, we take the weighted sum of all the units in the first layer, according to the Θ^1 parameter, and then add the bias value. Then we run the activation function on this sum, to get the activated value for this unit.
Similiarly, we can calculate the value for all units in the second layer. Using these values, we can then calculate the values for the third layer, and so on till the last layer.

1.2. Back-Propagation

When we compare our output layer with the ground truth, we get the value of δ^L, which gives a measure of how incorrect our prediction is, compared to the ground truth.
Now to decrease this delta, each unit in the output layer will want some changes in the penultimate layer. Taking all of these changes into account, we get an intuition of the changes required for the second-last

layer, i.e. δ^{L-1}.
Now by similiar methods, we can find the δ for each unit of each layer, and then finally we can use that to find the changes required in the weights matrices between each consecutive pair of layers.
It is called back-propagation, as we start by calculating the error in the output layer, and move backward towards the first layer to find all the errors.

2. Question 2

Assumptions:

1. Number of units in each layer taken according to diagram (i.e. num_input = 4, num_hidden = 5, num_output = 1)
2. Regularization not performed
3. All relevant matrices and units have been previously defined

2.1. Without Vectorisation

Without vectorisation, we have to use for-loops to calculate the values required.

```
# FORWARD PROPAGATION
a = [0] * 5 # a is hidden layer
for i in range(5):
    for j in range(4):
        a[i] += weights_1[i][j] * x[j]
    a[i] = sigmoid(a[i] + bias_1)
h = 0 # h is hypothesis, i.e. output layer
for i in range(5):
    h += weights_2[i] * a[i]
h = sigmoid(h + bias_2)
print(h)

# BACKWARD PROPAGATION

delta_3 = h - y
delta_2 = [0] * 5
for i in range(5):
    delta_2[i] = weights_2[i] * delta_3 *  (a[i] * (1-a[i]))
    Delta_2[i] += delta_3 * a[i]
for i in range(5):
    for j in range(4):
        Delta_1[i][j] += delta_2[j] * x[i]

bias_1_grad = delta_3
bias_2_grad = [d3 * t2 for d3, t2 in zip(delta_3, Theta_2)]
Theta_2_grad = [x/m for x in Delta_2]
Theta_1_grad = [[x/m for x in y] for y in Delta_1]
```

2.2. With Vectorisation

Vectorisation allows us to use matrix multiplication instead of for-loops. This allows us to make use the parallel processing power present in most recent computers.

```
# FORWARD PROPAGATION

a = sigmoid(x * weights_1.T + bias_1)
h = sigmoid(a * weights_2.T + bias_2)
print(h)

# BACKWARD PROPAGATION

delta_3 = h - y
delta_2 = np.multiply(delta3 * weights_2, sigmoidGradient(z2))

Delta_2 = delta_3 * a
Delta_1 = delta_2 * x

bias_2_grad = delta_3
bias_1_grad = delta_3 * Theta2
Theta_1_grad = (1/m) * Delta_1
Theta_2_grad = (1/m) * Delta_2
```

3. Question 2 (BONUS)

The vectorised implementation above is independent of the size of any layers, and is therefore can be applied to any basic neural network, with one layer.

4. Question 3

4.1. Sigmoid

Activation Function:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Derivative:

$$\begin{aligned} f'(x) &= f(x) * (1 - f(x)) \\ &= \frac{e^{-x}}{(1 + e^{-x})^2} \end{aligned}$$

4.2. ReLU

ReLU is an activation function, designed with strong biological motivations and mathematical justifications. It is the most common activation function used for activating the hidden layers.

Activation Function:

$$f(x) = x^+ = \begin{cases} x & \text{if } x > 0 \\ 0 & \text{if } x <= 0 \end{cases}$$

Derivative:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x < 0 \end{cases}$$

The derivative of ReLU is undefined at x = 0, which is a potential problem.

4.3. Leaky ReLU

Activation Function:

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.01x & \text{if } x <= 0 \end{cases}$$

Derivative:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0.01 & \text{if } x <= 0 \end{cases}$$

The derivative of a leaky ReLU is also undefined at x = 0.
However, the addition of the slight positive gradient at x < 0 can prevent the **Dying ReLU** problem.

4.4. tanh (Hyperbolic Tangent)

Activation Function:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Derivative:

$$(\tanh x)' = \left(\frac{\sinh x}{\cosh x}\right)' = \frac{(\sinh x)'\cosh x - \sinh x(\cosh x)'}{\cosh^2 x} = \frac{\cosh x \cdot \cosh x - \sinh x \cdot \sinh x}{\cosh^2 x}$$

$$= \frac{\cosh^2 x - \sinh^2 x}{\cosh^2 x} = \frac{1}{\cosh^2 x} = \operatorname{sech}^2 x.$$

$$\text{Where} : \operatorname{sech}(\mathrm{x}) = \frac{2e^x}{e^{2x} + 1}$$

4.5. Softmax

The softmax function takes a vector of K reals, and normalises them into a probability distribution.
The probability of each case is proportional to the exponential of the number, and normalisation ensures that the total sum of all probabilities is 1. THerefore this is mostl used as the activation function for the output layer.

Activation Function:

$$\sigma(z_i) = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$$

Here β is usually taken as 1, in the standard softmax function.
Chosing larger values for β creates a probability distribution that is more concentrated around the positions of the larger inputs.

Derivative:

Since the function maps a vector and a specific index *i* to a real value, the derivative needs to take the index into account

$$\frac{\partial}{\partial q_k}\sigma(q, i) = \sigma(q, i)(\delta_{ik} - \sigma(q, i))$$

Here δ_{ik} is the kronecker-delta, which returns 1 if i = k, and 0 otherwise.
Therefore the above derivative is symmetric wrt. i, and k