UNIVERSITY COLLEGE LONDON
DEPARTMENT OF SPACE AND CLIMATE PHYSICS

Candidate Code: HYXC3

Programme Title: MSc Scientific Computing

Module Code: SPCE0038

Module Title: Machine Learning with Big Data

End Assessment

Date of Submission:	11/05/20

Question 1

1(a)

With reference to the diagram of the basic *logistic unit* on the following page:

The input vector $\mathbf{x}$ is the input to the *logistic unit.*

Each input x_i has an associated weight θ_i. The weights are set to a random value prior to *training.* The *training* process determines these weights.

The product of each input x_i and weight θ_i is summed to produce a weighted sum z.

The output, $h_\theta(x)$, is the the non-linear *activation function*, h, applied to z.

1(b)

Consider the diagram of Question 1(a).

The weighted sum of the inputs, z, is:

$$z = \sum_{j=1}^{n} \theta_j x_j = \theta^T x \tag{1}$$

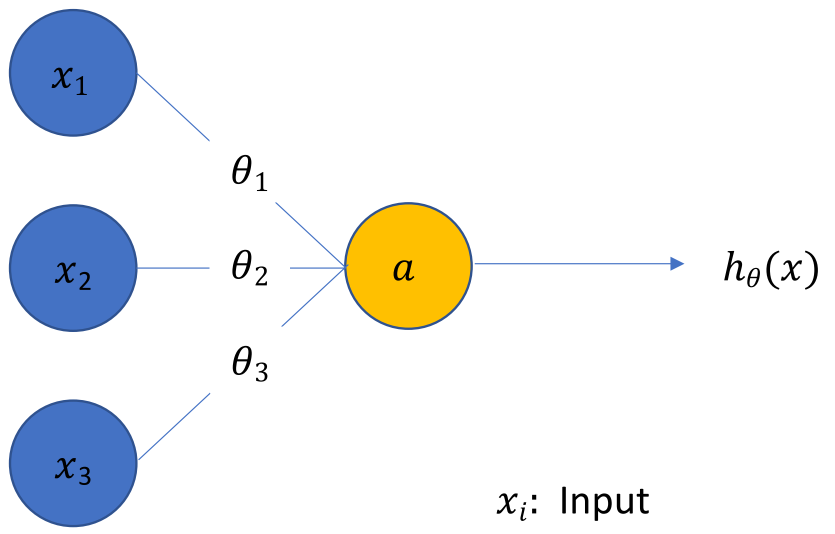
where x_i is the i^{th} element of input vector $\mathbf{x}$ of length n, and θ_i is the associated *weight.*

And, the output from the *logistic unit*, $h_\theta(x)$, is:

$$h_\theta(x) = h(z) \tag{2}$$

where h is a non-linear *activation function.*

Question 1(a) – Basic Logistic Unit



x_i: Input

θ_i: Weight

Weighted Sum: $z = \sum_{j=1}^{n} \theta_j x_j = \theta^T x$

Activation: $a = h(z)$

1(c)

See diagram on following page.

1(d)

Consider the diagram of Question 1(c).

Firstly, consider the data transformation from the input vector $\mathbf{x}$ to the hidden layer. We now need two indices, one for the input vector elements, and one for the *hidden layer* nodes. We will use i for the input vector index, and j for the *hidden layer* nodes.

The weighted sum of the inputs at the j^{th} *hidden layer* node is:

$$z_j = \sum_{i=1}^{n} \theta_{ij} x_i \tag{3}$$

where θ_{ij} is the weight between input element i and *hidden layer* node j, and n is the length of the input vector $\mathbf{x}$.

Secondly, the output from each *hidden layer* node is the non-linear *activation function*, h, applied to each z_j:

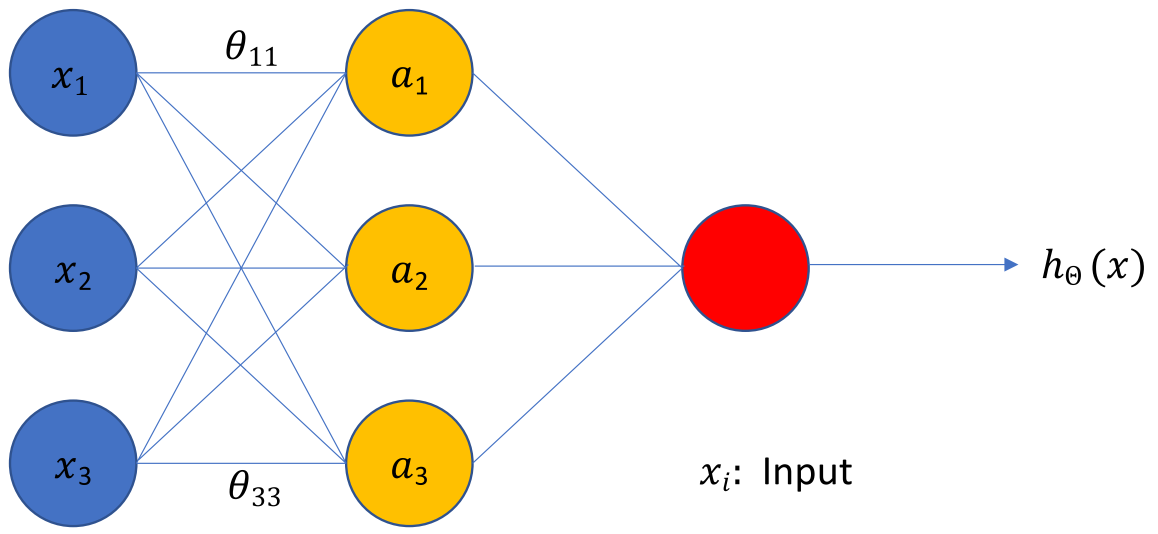
$$h_{\theta j}(x) = h(z_j) \tag{4}$$

And finally, the output from the whole network, $h_\Theta(x)$ is the sum of the *hidden layer* outputs:

$$h_\Theta(x) = \sum_{j=1}^{m} h_{\theta j}(x) \tag{5}$$

where m is the number of *hidden layer* nodes.

Question 1(c) – Fully Connected, Feed Forward, Artificial Neural Network



x_i: Input

θ_{ij}: Weight, e.g. $\theta_{11}\,\theta_{33}$

Weighted Sums: $z_j = \sum_{i=1}^{n} \theta_{ij} x_i$

Activations: $a_j = h(z_j)$

🔵 Input Layer Logistic Units

🟡 Hidden Layer Logistic Units

🔴 Output Node

1(e)

The cost function typically used to train neural networks for regression problems is *mean square error*:

$$MSE(\Theta) = \frac{1}{m} \sum_i \sum_j (p_j^{(i)} - y_j^{(i)})^2 \tag{6}$$

The cost function typically used to train neural networks for classification problems is *cross-entropy*:

$$C(\Theta) = -\frac{1}{m} \sum_i \sum_j y_j^{(i)} \log(p_j^{(i)}) \tag{7}$$

1(f)

Artificial Neural Networks (ANNs) are described as *shallow* or *deep*, and *wide* or *narrow*. *shallow* or *deep* refers to the number of layers in the network, and *wide* or *narrow* refers to the number of nodes in each layer.

The *credit assignment path*, the CAP, of a neural network is a measure of the number of data transformations that occur as data passes through the network. For *feed-forward* networks the CAP is the number of *hidden layers* plus one.

A *deep* neural network is generally considered to be a network with multiple layers and a CAP > 2.

1(g)

The *universal approximation theorem* states that, with appropriate parameters, single hidden layer feed-forward neural networks are *universal approximators*. This means they can represent any continuous function. However, this requires an exponentially larger number of hidden layer nodes. And, training will not necessarily determine the parameters.

Deep networks provide a powerful representational framework because they have the potential to be *universal approximators*, but with a limited width of hidden nodes. This makes the implementation of *universal approximators* more feasible.

Question 2

2(a)

Gradient Descent algorithms attempt to find the parameters θ which minimise the cost function $C(\theta)$ over the *training set* using an iterative process:

$$\theta^{i+1} = \theta^i - \alpha \nabla_\theta C(\theta) \tag{8}$$

where α is the *learning rate*.

Batch Gradient Descent uses the entire *training set* at each iteration to calculate the gradient partial derivatives, $\nabla_\theta C(\theta)$. This produces accurate values for the partial derivatives, but is potentially slow for large *training sets*.

Stochastic Gradient Descent uses a random sub-set of the *training set* at each iteration to calculate the gradient partial derivatives, $\nabla_\theta C(\theta)$. This is faster than *Batch Gradient Descent*, but can produce erratic values for the partial derivatives.

For *convex* cost functions *Batch Gradient Descent* will always converge to the *local minima* which is also the *global minima*. This is not the case for *non-convex* cost functions, where *local minima* are not necessarily *global minima*. A bad choice of θ^0 may result in *Batch Gradient Descent* getting "stuck" in a *local minima*. Because the gradient partial derivatives of *Stochastic Gradient Descent* are erratic, this provides a mechanism of "jumping out of" a *local minima* and improves the probability of finding the *global minima*.

2(b)

When attempting to find the minimum of a cost function using *Stochastic Gradient Descent*, the iterative process "jumps" around the minimum, and it is difficult to determine when a minimum has been reached. For this reason alternative optimisation algorithms are typically considered for training.

2(c)

Consider the iterative minimisation process:

$$\theta^{i+1} = \theta^i - \alpha \nabla_\theta C(\theta) \tag{9}$$

where α is the *learning rate*.

At each step the process "advances" towards the minimum by the step size $-\alpha \nabla_\theta C(\theta)$, which uses the *current* gradient.

The *Momentum Optimisation Algorithm* introduces the idea of including *previous* gradients into the step size. At each step the *current* gradient is summed into a momentum term, m (remember we use the negative gradient):

$$m^{i+1} = \beta m^i - \alpha \nabla_\theta C(\theta) \tag{10}$$

The β parameter is used to prevent the momentum getting too large, and is set between 0 and 1, typically 0.9.

The momentum m is then used to update θ:

$$\theta^{i+1} = \theta^i + m \tag{11}$$

The result is that we now have an *acceleration* towards the minimum.

2(d)

2(e)

2(f)

2(g)

2(h)

Question 3

3(a)

TODO

3(b)

TODO

3(c)

TODO

3(d)

TODO

3(e)

TODO

3(f)

TODO

Question 4

4(a)

TODO

4(b)

TODO

4(c)

TODO

4(d)

TODO

4(e)

TODO

4(f)

TODO

question_4f

May 7, 2020

[ ]:
```python
# Fetch batch function:

def fetch_batch(epoch, batch_index, batch_size):

    return X_batch, y_batch


# Set up computational graph:

import tensorflow as tf
reset_graph ()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")

theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf .matmul(X, theta , name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(mse)


# Execute:

init = tf.global_variables_initializer()

with
tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE=", mse.eval()) sess.run(training_op)
    best_theta = theta.eval()
```

```python
# Fetch batch function:

def fetch_batch(epoch, batch_index, batch_size):
    return X_batch, y_batch


# Set up computational graph:

import tensorflow as tf
reset_graph ()

n_epochs = 1000
learning_rate = 0.01

X = tf.constant(scaled_housing_data_plus_bias, dtype=tf.float32, name="X")
y = tf.constant(housing_data_target, dtype=tf.float32, name="y")

theta = tf.Variable(tf.random_uniform([n + 1, 1], -1.0, 1.0), name="theta")
y_pred = tf .matmul(X, theta , name="predictions")
error = y_pred - y
mse = tf.reduce_mean(tf.square(error), name="mse")
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
training_op = optimizer.minimize(mse)


# Execute:

init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    for epoch in range(n_epochs):
        if epoch % 100 == 0:
            print("Epoch", epoch, "MSE=", mse.eval())
            sess.run(training_op)
    best_theta = theta.eval()
```

Listing 1: Question 4f

Question 5

5(a)

TODO

5(b)

TODO

5(c)

TODO

5(d)

TODO

5(e)

TODO

5(f)

TODO