Annotate code

Consider a fully connected network constructed using the `__init__` method given below.

Summary of Feed Forward computation:

$z^l = w^l a^{l-1} + b^l \;\; a^l = \sigma(z_l)$

We assumed Sigmoid Non Linearity in our example

In [ ]:

```
import numpy as np

class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)  # Number of layers in our neural network including the input layer
        self.sizes = sizes            # Storing the sizes list in object variable to use it other methods in
                                      # this class
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]  # We need to choose inital parameter numbers
                                      # to compute the forward step and these weights will be in the gradient des
cent step
                                      # From the Equation 1, we can see that b_l is a vector and it should be of
size number of neurons in the layer l. Another point to note is that index 0 in sizes list
                                      # represents the input layer and the weights and biases are defined from th
e hidden layer 1. That is the why we are iterating from index 1 from the sizes list
        self.weights = [np.random.randn(y, x) # From the Equation 1, it is clear that Weights w should be of size
num(previous_layer_neurons) * num(current_layer_neurons). and these weights are initialzed
                                              # by sampling from the Standard Normal distribution (Mean =0, Varie
nce =1)
                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights): # Iterating over all the layers to compute the feedforward st
ep in neural network in a recursive way.
            a = sigmoid(np.dot(w, a)+b) # We are applying first the equation 1 : z_l = w_l * a_l-1 + b_l and the
Equation 2: a_l = sigmoid(z_l)
        return a
```

- Comment every code line in `backprop` with the analytical expression that line evaluates in computing ∇C
- Schematically apply `backprop` on a network constructed as `net = Network([784, 30, 10])`

Back Propagation Equations

Summary: The equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma^{'}(z^L)$$

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma^{'}(z^l)$$

Using δ_L (Last Layer) and δ_l (For the remaining layer) we compted the gradients with respect to the Weights and Biases in each layer

$$\frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$$

Copmutation of $\nabla_a C$ is shown below

- Cost for a single training example is $C_x = \frac{1}{2} ||y - a_L||^2$

$\frac{\partial C}{\partial a_L} = (y - a_L)(-1) = (a_L - y)$

- From the notation $\nabla_a C = \frac{\partial C}{\partial a_L} = (a_L - y)$

The above relation 7 is directly used in the Equation 1 of summary of backpropgation steps.

- *Note: This formula will change depending on the choice of loss function*

In [1]:

```
def backprop(self, x, y):
    """Return a tuple "(nabla_b, nabla_w)" representing the
    gradient for the cost function C_x.  "nabla_b" and
    "nabla_w" are layer-by-layer lists of numpy arrays, similar
    to "self.biases" and "self.weights"."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]  # Since gradient is computed for each parameter, we are
                                                        #initializing the gradients of each bias parameter to be
zero
                                                        #with the same number of entries as biases parameters
    nabla_w = [np.zeros(w.shape) for w in self.weights] # Since gradient is computed for each parameter, we are
                                                        #initializing the gradients of each weight parameters to
be zero
                                                        #with the same number of entries as weight parameters
    # feedforward
    activation = x                                      # Activation values "a" for the first layer is equal to i
nput "x" itself
    activations = [x]                                   # list to store all the activations, layer by layer
                                                        # We need to store all the intermediate activations to co
mpute the backpropgation updates
    zs = []                                             # list to store all the z vectors, layer by layer (zs var
iable represent before the sigmoid function is applied)
                                                        # all the intermediate Linear transformation values are a
lso required for the backpropagation steps

    for b, w in zip(self.biases, self.weights):         # Iterating over all the layers to compute the feedforwar
d step in neural network in a recursive way.
        z = np.dot(w, activation)+b                     # computation of linear tranformation function z = Wx+b.
where x is the activation map from the previous layers. np.dot is matrix multiplcation
        zs.append(z)                                    # Storing the computations in a list to use it in the bac
kpropagation step. In particualy we need these values to
                                                        # compuate the derivative of sigmoid at these values. Equ
ation 1 and 2 from the summary of backpropagation equations
        activation = sigmoid(z)                         # Applying sigmoid non-leariy function f(x) = 1/(1 + np
.exp(-x)) element wise. (Numpy applies element wise to vectore quantity)
        activations.append(activation)                  # Storing the intermediate activation maps for the backpr
opagation steps. Equation 4 requires these values to compute
                                                        # the partial derivate of Loss with respect to weights.
    # backward pass
    delta = (activations[-1] - y) * sigmoid_prime(zs[-1])  # This step computes the derivative of Loss with respe
ct to "z variables" in the last layer of neural network.
                                                           # Since we assumed to use squared loss function (a_i-y
_i)^2 / 2, derivative of Loss with respect to output activation
                                                           # values is (a_i - y_i) and the derivative of Loss wit
h respect to "z variables" is product of
                                                           #derivative of Loss with respect to "output activation
" * derivative of sigmoid function computed at the z variable
                                                           # Detailed computation of this step is provided in the
above section of this code (Equations 5, 6, 7)
    nabla_b[-1] = delta                                    # Using Equation 3, we are computng the gradient with
to "b variables" in the last layer of neural network.
    nabla_w[-1] = np.dot(delta, activations[-2].transpose()) #Using Equation 4, we are computng the gradient with
to "w variables" in the last layer of neural network.
    for l in xrange(2, self.num_layers):                 # Now, we are applying the recursive backpropagation step.
Since we already computed the delta values for the last layer, we use this last layer delta values and
                                                         # Compute the delta for the previous layers.
        z = zs[-l]                                       # From Equation 2, inorder to apply compte delta values re
cursively, we need to computed the derivative of sigmoid at the "z variables"
        sp = sigmoid_prime(z)                            # 3rd term in Equation 2 RHS, requires the derivative of s
igmoid at the "z variables"
        delta = np.dot(self.weights[-l+1].transpose(), delta) * sp  # Computing the delta values recursively usin
g equation 2 : delta_l = W^(l+1).T * delta_l+1 * derivative of sigma
        nabla_b[-l] = delta                              # Using Equation 3 : C/ partial b = delta, we are computng
the gradient with to "b variables" in the current layer of neural network. partial
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose()) #Using Equation 4: partial C/ partial W = delt
a * a.T, we are computng the gradient with to "w variables" in the current layer of neural network.
    return (nabla_b, nabla_w)
```

II Using matrix notation

Forward Step

Summary of Feed Forward computation:

$z^l = w^l a^{l-1} + b^l \;\; a^l = \sigma(z^l)$

Let us compute the z values for the first layer: $z^1 = w^1 a^0 + b^1$ and $a^1 = \sigma(z_1)$

Since a^0 is equal to input x Equation 1 becomes $z^1 = w^1 x + b^1$

Lets process the input as a mini-batch of size m. Take all the inputs for all the datapoints from $1, 2, 3....m$ and stack the vectors along the column. Now the data matrix becomes

$$X = \begin{bmatrix} | & | & | & | & | & | \\ x_1 & x_2 & . & . & . & x_m \\ | & | & | & | & | & | \end{bmatrix}$$

where x_i representing input vector of i^{th} instance

From the Equation $z^1 = w^1 x + b^1$ replacing x with X results in the following expression

$z^1 = w^1 X + b^1$

Expanding the above expression and use the matrix matrix product with one column at at time

$$z^1 = w^1 \begin{bmatrix} | & | & | & | & | & | \\ x_1 & x_2 & . & . & . & x_m \\ | & | & | & | & | & | \end{bmatrix} + b^1$$

$$z^1 = \begin{bmatrix} | & | & | & | & | & | \\ w^1 x_1 & w^1 x_2 & . & . & . & w^1 x_m \\ | & | & | & | & | & | \end{bmatrix} + b^1$$

Observing the equation 7, Each column in the resulting matrix correpsonds to genuine computation of $w^1 x_i$ and repeating the columns of vector b^1, m times this will exactly corresponds to forward computation of our deep neural network model

In Summary: By Stacking the input data along the column dimensing and repeating the columns of b^1 vector total m times (mini-batch size), we can directly compute $z^1 = w^1 X + b^1$ and each column in z^1 matrix corresponds to one data instance. There is no change to be done during the forward step in the code. Numpy broadcasting operation also helps in repeating the bias vectors to match the mini-batch dimension

Note:In Python We dont have to repeat the columns of b^1, since numpy has broadcasting operation

Similary, since we are applying sigmoid function element-wise to the "z" variable. Each column in a^1 matrix corresponds to one data instance

Conclusion: When we modify the input x with X containing all the mini-batch instances stacked along the column dimension, all the intermediate z^l variables and a^l variables and the last layer z^L , a^L are matrices and each i^{th} column in these matrices corresponds to i^{th} data instance

Backpropagation equations

$\nabla_a C = \frac{\partial C}{\partial a_L} = (a_L - y)$

$$\delta^L = \nabla_a C \odot \sigma'(z^L)$$

$$\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma'(z^l)$$

Using the error terms we compted the gradients with respect to the Weights and Biases in each layer

$$\frac{\partial C}{\partial b^l} = \delta^l$$

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$$

Analyzing Equation 1: $\nabla_a C = \frac{\partial C}{\partial a_L} = (a_L - y)$

- From the above argument when we represent input data with $X = \begin{bmatrix} | & | & | & | & | & | \\ x_1 & x_2 & . & . & . & x_m \\ | & | & | & | & | & | \end{bmatrix}$ then the activation matrix at the last layer becomes

  $a_L = \begin{bmatrix} | & | & | & | & | & | \\ a_{L1} & a_{L2} & . & . & . & a_{Lm} \\ | & | & | & | & | & | \end{bmatrix}$ where a_{Li} represents activation for the i^{th} data sample.

- Similarly if we encode the output as input $Y = \begin{bmatrix} | & | & | & | & | & | \\ y_1 & y_2 & . & . & . & y_m \\ | & | & | & | & | & | \end{bmatrix}$ then i^{th} column in $a_L - Y$ matrix corresponds to i^{th} data instance in mini-batch

Analyzing Equation 2: $\delta^L = \nabla_a C \odot \sigma^{'}(z^L)$

- Since this equation $\delta^L = \nabla_a C \odot \sigma^{'}(z^L)$ involves element wise multiplication. Each column (i^{th})in the matrix δ^L corresponds to i^{th} data instance in mini-batch

Analyzing Equation 3: $\delta^l = (W^{l+1})^T \delta^{l+1} \odot \sigma^{'}(z^l)$

- When we process the mini-batch of data then each columns in the matrix δ^L corresponds to i^{th} example (from the above argument). Expanding δ^{l+1} in terms of each columns $\delta^{l+1} = \begin{bmatrix} | & | & | & | & | & | \\ \delta^{l+1}_1 & \delta^{l+1}_2 & . & . & . & \delta^{l+1}_m \\ | & | & | & | & | & | \end{bmatrix}$ and $\sigma^{'}(z^l) = \begin{bmatrix} | & | & | & | & | & | \\ \sigma^{'}(z^l_1) & \sigma^{'}(z^l_2) & . & . & . & \sigma^{'}(z^l_m) \\ | & | & | & | & | & | \end{bmatrix}$

- Then Equation 3 becomes $\delta^l = (W^{l+1})^T \begin{bmatrix} | & | & | & | & | & | \\ \delta^{l+1}_1 & \delta^{l+1}_2 & . & . & . & \delta^{l+1}_m \\ | & | & | & | & | & | \end{bmatrix} \odot \begin{bmatrix} | & | & | & | & | & | \\ \sigma^{'}(z^l_1) & \sigma^{'}(z^l_2) & . & . & . & \sigma^{'}(z^l_m) \\ | & | & | & | & | & | \end{bmatrix}$

- Using the Property $A \begin{bmatrix} | & | & | & | & | & | \\ col_1 & col_2 & . & . & . & col_m \\ | & | & | & | & | & | \end{bmatrix} = \begin{bmatrix} | & | & | & | & | & | \\ Acol_1 & Acol_2 & . & . & . & Acol_m \\ | & | & | & | & | & | \end{bmatrix}$
- Equation 3 Simplifies to

$$\delta^l = \begin{bmatrix} | & | & | & | & | & | \\ (W^{l+1})^T \delta^{l+1}_1 \odot \sigma^{'}(z^l_1) & (W^{l+1})^T \delta^{l+1}_2 \odot \sigma^{'}(z^l_2) & . & . & . & (W^{l+1})^T \delta^{l+1}_m \odot \sigma^{'}(z^l_m) \\ | & | & | & | & | & | \end{bmatrix}$$

- *Looking at the above equation it is clear that i^{th} column in δ^l corresponds to i^{th} data point in mini-batch*

Analyzing Equation 4: $\frac{\partial C}{\partial b^l} = \delta^l$

- Since $\delta^l = \begin{bmatrix} | & | & | & | & | & | \\ \delta^l_1 & \delta^l_2 & . & . & . & \delta^l_m \\ | & | & | & | & | & | \end{bmatrix}$ and each columns values corresponds to i^{th} data instance(From above argument)

To compute mini-batch gradient $\frac{\partial C}{\partial b^l} = \frac{\sum_{i=1}^m \frac{\partial C}{\partial b^l_i}}{m} = \frac{\sum_{i=1}^m \delta^l_i}{m}$

In order to compute the gradient update for the mini-batch data, we need to take the average of all columns vectores in δ^l matrix

```
np.sum(axis=0)
```

Analyzing Equation 5: $\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T$

- Since $\delta^l = \begin{bmatrix} | & | & | & | & | & | \\ \delta^l_1 & \delta^l_2 & . & . & . & \delta^l_m \\ | & | & | & | & | & | \end{bmatrix}$ and $a^{l-1} = \begin{bmatrix} | & | & | & | & | & | \\ a^{l-1}_1 & a^{l-1}_2 & . & . & . & a^{l-1}_m \\ | & | & | & | & | & | \end{bmatrix}$
- The product $\delta^l (a^{l-1})^T$ becomes $\delta^l_1 (a^{l-1}_1)^T + \delta^l_2 (a^{l-1}_2)^T + . + . + . + \delta^l_m (a^{l-1}_m)^T$

Each quantity $\delta^l_i (a^{l-1}_i)^T$ in the summation represents a matrix for the i^{th} instance. Since gradeint for the entire mini-batch involves summation for all the data instaces, the product $\delta^l (a^{l-1})^T$ capures the summation information

To compute mini-batch gradient $\frac{\partial C}{\partial w^l} = \frac{\sum_{i=1}^m \frac{\partial C}{\partial w^l_i}}{m} = \frac{\sum_{i=1}^m \delta^l_i (a^{l-1}_i)^T}{m}$

- From the Equation 10: Numerator of this expression is equal given by $\delta^l (a^{l-1})^T$

In Summary, In order to copmute the gradient update of weights for the mini-batch data, we need to divide the matrix $\delta^l (a^{l-1})^T$ with mini-batch size (m)

The conclusion of the above mathematical analysis are the following:

- Assuming the data poinsts x_i and y_i are stacked along the column to make X and Y
- Using the numpy broadcasting operation during the fowards pass
- We have to make change only in two places
  - Computing the average of column vectors in δ^l matrix
  - Diving the matrix $\delta^l (a^{l-1})^T$ with the batch-size m

The result of this gives gradients with respect to weights and biases for the mini-batch data with out for loops

In [13]:

```
import numpy as np
import random

class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]


        self.weights = [np.random.randn(y, x)

                        for x, y in zip(sizes[:-1], sizes[1:])]

    def feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a


    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x.  "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        m = len(x)
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = (activations[-1] - y) * sigmoid_prime(zs[-1])
        nabla_b[-1] = delta         # From the above mathematical anaysis, it is
                                    # clear that we need to take the average along the column dimension
        nabla_b[-1] = np.mean(nabla_b[-1], axis=1)  #( takeing average and axis=1 represents along the column dim
ension)
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
                                    # From the above mathematical analysis, it is clear that we need to divide th
e nabla_w with the batch size
        nabla_w[-1] = nabla_w[-1] / m
```

```
        for l in range(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta   # From the above mathematical anaysis, it is
                                    # clear that we need to take the average along the column dimension
            nabla_b[-l] = np.mean(nabla_b[-l], axis=1) #( takeing average and axis=1 represents along the column
dimension)
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
                                  # From the above mathematical analysis, it is clear that we need to divide the
nabla_w with the batch size

            nabla_w[-l] = nabla_w[-l] /m
        return (nabla_b, nabla_w)

    def SGD(self, training_data, epochs, mini_batch_size, eta):
        """Train the neural network using mini-batch stochastic
        gradient descent.  The ``training_data`` is a list of tuples
        ``(x, y)`` representing the training inputs and the desired
        outputs.  The other non-optional parameters are
        self-explanatory.  """
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)


    def update_mini_batch(self, mini_batch, eta):
        """Update the network's weights and biases by applying
        gradient descent using backpropagation to a single mini batch.
        The ``mini_batch`` is a list of tuples ``(x, y)``, and ``eta``
        is the learning rate."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
#         for x, y in mini_batch:
#             delta_nabla_b, delta_nabla_w = self.backprop(x, y)
#             nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
#             nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
        X,Y = mini_batch                      # Storing the entire batch of data into X and Y matrices.
        nabla_b , nabla_w = self.backprop(X,Y) # backprop takes X, Y and return the average of gradients
                                               # of all the images in a batch without for loop


#         self.weights = [w-(eta/len(mini_batch))*nw
#                         for w, nw in zip(self.weights, nabla_w)]
#         self.biases = [b-(eta/len(mini_batch))*nb
#                        for b, nb in zip(self.biases, nabla_b)]

        self.weights = [w-(eta)*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta)*nb
                       for b, nb in zip(self.biases, nabla_b)]

        # I removed the denominator len(mini_batch) since we are getting averages from the backprop function

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))



def sigmoid_prime(z):
    return sigmoid(z)*(1-sigmoid(z))
```

In [14]:

```
net = Network([784,30,10])
```

In [15]:

```
net.feedforward(np.random.randn(784,1))
```

Out[15]:

```
array([[3.97552067e-03],
       [7.22695924e-04],
       [9.96692507e-01],
       [8.26339636e-01],
       [1.10037233e-02],
       [9.96979140e-01],
       [2.68942117e-01],
       [1.67113032e-04],
       [6.27497188e-04],
       [9.05265013e-01]])
```

In [16]:

```
X= np.random.randn(784,1000)
Y= np.random.randn(10,1000)
```

In [17]:

```
nabla_b2,nabla_w2 = net.backprop(X,Y)
```

In [18]:

```
nabla_w2[-1].shape
```

Out[18]:

```
(10, 30)
```

In [19]:

```
nabla_b2[-1].shape
```

Out[19]:

```
(10,)
```

In [ ]:

```
x1= X[:,0]
x1=x1.reshape(784,1)
print(x1.shape)
y1= Y[:,0]
y1=y1.reshape(10,1)
print(y1.shape)
```

```
(784, 1)
(10, 1)
```

In [ ]:

```
y1.shape
```

Out[ ]:

```
(10, 1)
```

In [ ]:

```
nabla_b,nabla_w = net.backprop(x1,y1)
```

In [ ]:

```
nabla_b[-2]
```

In [ ]:

```
nabla_w = np.zeros(shape=(10,30))
for i in range(1000):

    x1= X[:,i]
    x1=x1.reshape(784,1)
    y1= Y[:,i]
    y1=y1.reshape(10,1)
    a,b = net.backprop(x1,y1)
    nabla_w = nabla_w + b[-1]
```

In [ ]:

```
nabla_w
```

In [ ]:

```
nabla_w2[-1]
```

In [ ]:

```
! jupyter nbconvert --to html '/content/drive/MyDrive/Colab Notebooks/Assignment_1.ipynb'
```

```
[NbConvertApp] Converting notebook /content/drive/MyDrive/Colab Notebooks/Assignment_1.ipynb to html
[NbConvertApp] Writing 338412 bytes to /content/drive/MyDrive/Colab Notebooks/Assignment_1.html
```

In [ ]: