CSC411: Project #2

Due on Sunday, February 18, 2018

Hao Zhang & Tyler Gamvrelis

February 20, 2018

Foreword

In this project, neural networks of various depth were used to recognize handwritten digits and faces.

Part 1 provides a description of the MNIST dataset from which the images of handwritten digits were taken. Part 2 implements the computation of a simple network. Part 3 presents the *sum of the negative log-probabilities* as a cost function, and derives the expression for its gradient with respect to one of its weights. Part 4 details the training and optimization procedures for a digit-recognition neural network, and part 5 improves upon the results by modifying gradient descent to use momentum. Learning curves are presents in parts 4 and 5. Part 6 presents an analysis of network behaviour with respect to two of its weights. Part 7 provides an analysis of the performance of two different backpropagation computation techniques. Part 8 presents a face recognition network architecture for classifying actors, and uses PyTorch for implementation. Part 9 presents visualizations of hidden unit weights relevant to two of the actors. Part 10 uses activations of AlexNet to train a neural network to perform classification of the actors.

System Details for Reproducibility:

- Python 2.7.14
- Libraries:
  - numpy
  - matplotlib
  - pylab
  - time
  - os
  - scipy
  - urllib
  - cPickle
  - PyTorch

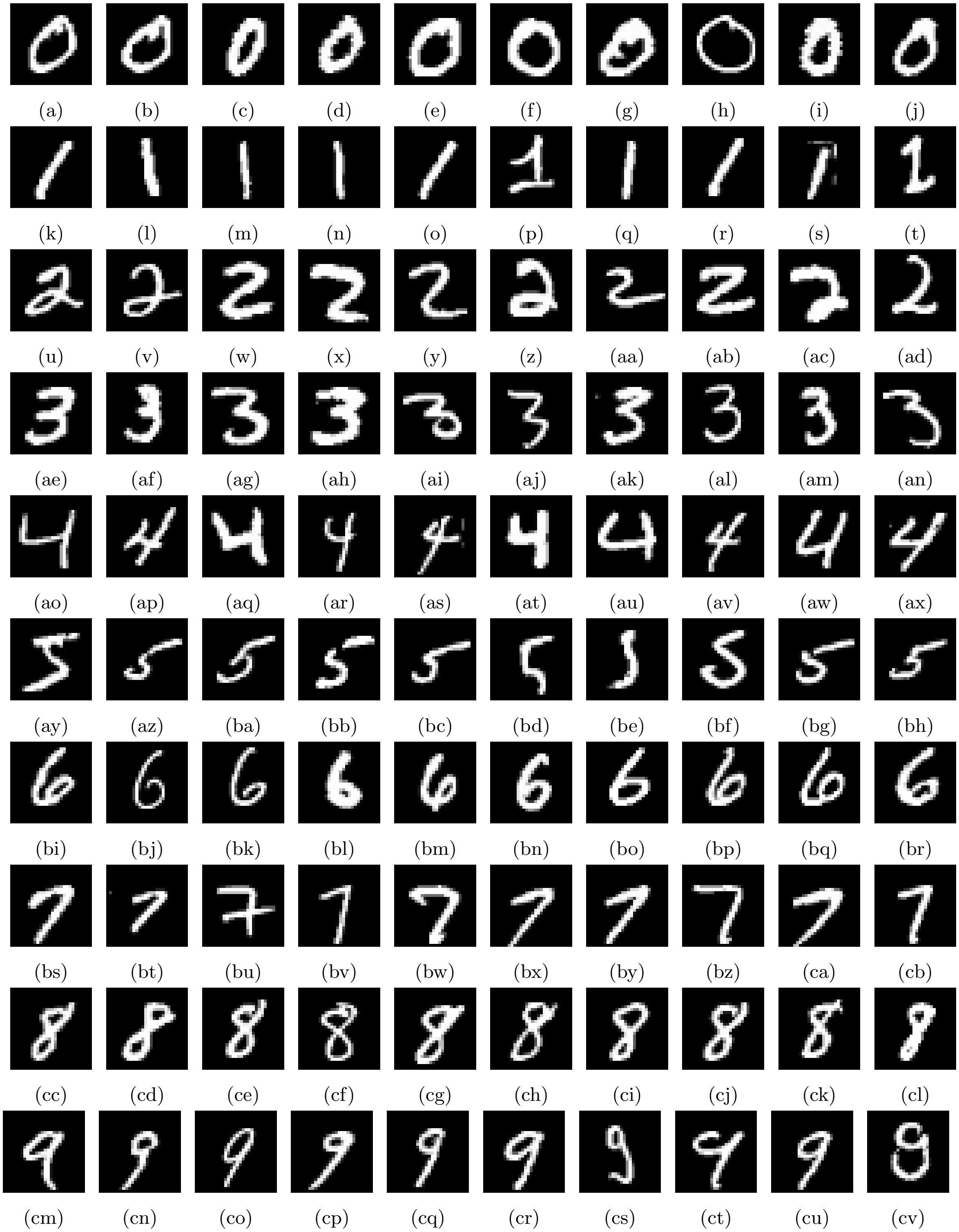
Part 1

Dataset description
The MNIST dataset is made of thousands of 28 by 28 pixel images of the handwritten digits: 0 to 9. The images are split into training set and test set images labelled ‘train0’ to ‘train9’ and ‘test0’ to ‘test9’. The number of images with each label is presented in Table 1, below.

Label	**Number of Images**	**Label**	**Number of Images**
train0	5923	test0	980
train1	6742	test1	1135
train2	5958	test2	1032
train3	6131	test3	1010
train4	5842	test4	982
train5	5421	test5	892
train6	5918	test6	958
train7	6265	test7	1028
train8	5851	test8	974
train9	5949	test9	1009

Table 1: Quantity of each type of image in the MNIST dataset.

Ten images of each number were taken from the training sets and displayed in Figure 1. The correct labels of most of the pictures can be discerned at a glance by humans However, since the digits are handwritten, some of them may not be completely obvious. For example, Figure 1cv is categorized as a 9 but looks like an 8.



(a) (b) (c) (d) (e) (f) (g) (h) (i) (j)

(k) (l) (m) (n) (o) (p) (q) (r) (s) (t)

(u) (v) (w) (x) (y) (z) (aa) (ab) (ac) (ad)

(ae) (af) (ag) (ah) (ai) (aj) (ak) (al) (am) (an)

(ao) (ap) (aq) (ar) (as) (at) (au) (av) (aw) (ax)

(ay) (az) (ba) (bb) (bc) (bd) (be) (bf) (bg) (bh)

(bi) (bj) (bk) (bl) (bm) (bn) (bo) (bp) (bq) (br)

(bs) (bt) (bu) (bv) (bw) (bx) (by) (bz) (ca) (cb)

(cc) (cd) (ce) (cf) (cg) (ch) (ci) (cj) (ck) (cl)

(cm) (cn) (co) (cp) (cq) (cr) (cs) (ct) (cu) (cv)

Figure 1: Subset of the MNIST dataset.

Part 2

Computing a Simple Network

In this part, the simple network depicted in Figure 2 was implemented as a function in Python using NumPy, the code listing for which is presented is Figure 3.

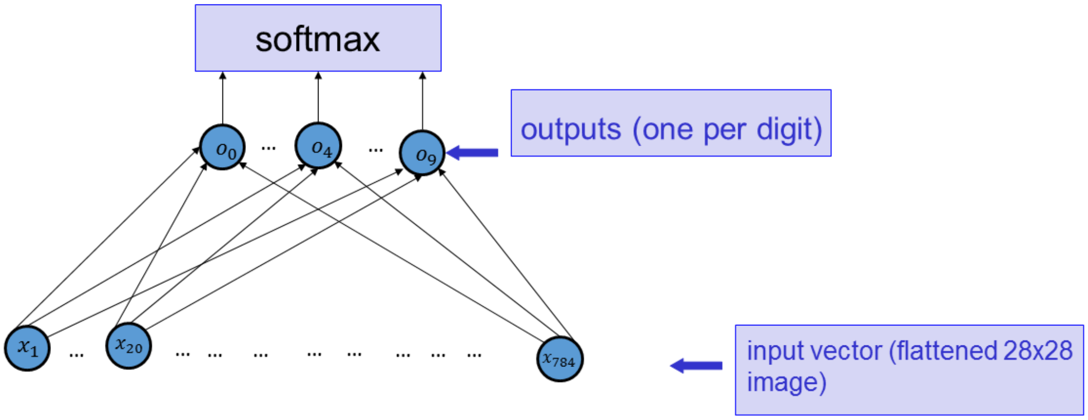


Figure 2: Simple network diagram from project handout.

```
def softmax(y):
     '''
     Return the output of the softmax function for the matrix of output y. y
     is an NxM matrix where N is the number of outputs for a single case, and M
     is the number of cases
     '''
     return exp(y)/tile(sum(exp(y),0), (len(y),1))

def SimpleNetwork(W, X):
    '''
     SimpleNetwork returns the vectorized multiplication of the (n x 10)
     parameter matrix W with the data X.
     Arguments:
          W -- (n x 10) matrix of parameters (weights and biases)
          x -- (n x m) matrix whose i-th column corresponds to the i-th training
     image
     '''
     return softmax(np.dot(W.T, X))
```

Figure 3: Python implementation of network using NumPy.

Part 3

Cost Function of Negative Log Probabilities
The Cost function that will be used for the network described in part 2 is:

$$C = -\sum_{q=1}^{m}(\sum_{l=1}^{k} y_l log(p_l))_q$$

Where k is the number of output nodes, m is the number of training samples, $p_l = \frac{e^{o_l}}{\sum_{c=1}^{k} e^{o_c}}$ and y_l is equal to 1 if the training sample is labelled as l and 0 otherwise.

Partial Derivative of the Cost Function (3a)
Let the matrix $W = \begin{bmatrix} w_{11} & w_{12} & w_{13} & \dots & w_{1k} \\ w_{21} & w_{22} & w_{23} & \dots & w_{2k} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ w_{n1} & w_{n2} & w_{n3} & \dots & w_{nk} \end{bmatrix}$ and let W_j be the j^{th} column of matrix W. This results in $o_j = W_j^T x$ where $x = \begin{bmatrix} x_1 \\ x_2 \\ \cdot\cdot \\ x_n \end{bmatrix}$ and n is equal to the number of pixels with an extra 1 to multiply into the bias. That is, x_1^q is always taken to be 1 while $x_2^q, ..., x_n^q$ are the inputs to the network corresponding to the q^{th} training example.

To calculate the partial derivative of C, we will consider only **a single training sample**: $C = -\sum_{l=1}^{k} y_l log(p_l)$
By the chain rule, the partial derivative with respect to w_{ij} is:

$$\frac{\partial C(W)}{\partial w_{ij}} = -\sum_{l=1}^{k} \frac{\partial C}{\partial p_l} \frac{\partial p_l}{\partial o_j} \frac{\partial o_j}{\partial w_{ij}}$$

The first term is straight forward: $\frac{\partial C}{\partial p_l} = \frac{y_l}{p_l}$.

For the second term, we have $\frac{\partial p_l}{\partial o_j} = \begin{cases} \frac{-e^{o_l}e^{o_j}}{(\sum_{c=1}^{k} e^{o_c})^2} = -p_l p_j, \text{if } l \neq j \\ \frac{e^{o_j}}{\sum_{c=1}^{k} e^{o_c}} - \frac{e^{2o_j}}{(\sum_{c=1}^{k} e^{o_c})^2} = p_j - p_j^2, \text{if } l = j \end{cases}$
From the equation: $o_j = W_j^T x$, we have $o_j = w_{1j}x_1 + w_{2j}x_2 + \cdots + w_{nj}x_n$ and it is clear to see that: $\frac{\partial o_j}{\partial w_{ij}} = x_i$.

Putting it all together, we have:

$$\frac{\partial C(W)}{\partial w_{ij}} = -(\sum_{l \neq j}(-\frac{y_l}{p_l} p_l p_j) + \frac{y_j}{p_j}(p_j - p_j^2))x_i = (\sum_{l \neq j}(y_l p_j) + y_j(p_j - 1))x_i$$

Considering that we are using 1-hot encoding, only a single y_l can equal 1. Therefore,

$$\frac{\partial C(W)}{\partial w_{ij}} = \begin{cases} (p_j - 1)x_i, & \text{if } y_j = 1 \\ p_j x_i, & \text{otherwise} \end{cases}$$

Which is equivalent to saying: $\frac{\partial C(W)}{\partial w_{ij}} = (p_j - y_j)x_i$. To extend this to all training samples, we just have to sum up each individual contribution, resulting in the equation: $\frac{\partial C(W)}{\partial w_{ij}} = \sum_{q=1}^{m}((p_j - y_j)x_i)_q$.

Vectorized Implementation of Gradient (3b)

To vectorize the gradient, we first define the gradient matrix to be:

$$\frac{\partial C(W)}{\partial W} = \begin{bmatrix} \frac{\partial C}{w_{11}} & \dots & \dots & \dots & \frac{\partial C}{w_{1k}} \\ \frac{\partial C}{w_{21}} & \dots & \dots & \dots & \frac{\partial C}{w_{2k}} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{\partial C}{w_{n1}} & \dots & \dots & \dots & \frac{\partial C}{w_{nk}} \end{bmatrix} = \begin{bmatrix} \sum_{q=1}^{m}((p_1 - y_1)x_1)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_1)_q \\ \sum_{q=1}^{m}((p_1 - y_1)x_2)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_2)_q \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \sum_{q=1}^{m}((p_1 - y_1)x_n)_q & \dots & \dots & \dots & \sum_{q=1}^{m}((p_k - y_k)x_n)_q \end{bmatrix}$$

This is equivalent to: $X(P - Y)^T$ where:

$$X = \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & x_1^{(3)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & x_2^{(3)} & \dots & x_2^{(m)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ x_n^{(1)} & x_n^{(2)} & x_n^{(3)} & \dots & x_n^{(m)} \end{bmatrix} Y = \begin{bmatrix} y_1^{(1)} & y_1^{(2)} & y_1^{(3)} & \dots & y_1^{(m)} \\ y_2^{(1)} & y_2^{(2)} & y_2^{(3)} & \dots & y_2^{(m)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ y_k^{(1)} & y_k^{(2)} & y_k^{(3)} & \dots & y_k^{(m)} \end{bmatrix} P = \begin{bmatrix} p_1^{(1)} & p_1^{(2)} & p_1^{(3)} & \dots & p_1^{(m)} \\ p_2^{(1)} & p_2^{(2)} & p_2^{(3)} & \dots & p_2^{(m)} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ p_k^{(1)} & p_k^{(2)} & p_k^{(3)} & \dots & p_k^{(m)} \end{bmatrix}$$

Where the superscript of each element represents the index of the training samples. The code used to compute the gradient is shown in Figure 4.

```python
def negLogLossGrad(X, Y, W):
    '''
    negLogLossGrad returns the gradient of the cost function of negative log
    probabilities.

    Arguments:
        X -- Input image(s) from which predictions will be made (n x m)
        Y -- Target output(s) (actual labels) (10 x m)
        W -- Weight matrix (n x 10)
    '''

    P = SimpleNetwork(W, X) # Get the prediction for X using the weights W

    return np.dot(X, (P - Y).transpose())
```

Figure 4: Vectorized Implementation of the Negative Log Loss Gradient

Finite Difference Approximation (3b)

To check that the gradient was computed correctly, the gradient with respect to weight w_{ij} was approximated using finite differences, for all coordinates in the weight matrix. 10 values for the differential quantity, h, were tested, for which the maximum total error was 2.698 for $h = 1$ and the minimum total error was 3.054×10^{-7} for $h = 1 \times 10^{-7}$. Thus, for sufficiently small h, the finite difference approximation and vectorized computation converge to the same quantity, suggesting the vectorized computation is correct. The test case used to obtain this result is shown in Figure 5. The complete results are summarized in Table 2, below.

Total error	**h**
2.698202291518717	1.0
0.200166359876926	0.1
0.019912135073950066	0.01
0.0019906216532900034	0.001
0.00019905646908402463	0.0001
1.990372311189148e-05	1e-05
1.98419058738963337e-06	1e-06
3.2450363651737035e-07	1e-07
3.0542104362263345e-06	1e-08
2.747131061797692e-05	1e-09

Table 2: Sum of differences between vectorized gradient computation and finite difference approximation for 10 h-values.

```python
# Test part 3
X = [[1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3, 4], [1, 2, 3 ,4],[1, 2, 3, 4]]
X = np.array(X)
W = [[1, 0, 1.2, 3], [1, 2, 0.2, 1.2], [1, 8, 1, 4], [1, -2, 3, 1], [1, 0, 3, 1]]
W = np.array(W)
Y = [[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
Y = np.array(Y)
G1 = negLogLossGrad(X, Y, W)
h = 1.0
for i in range(10):
    G2 = negLogLossGrad_FiniteDifference(X, Y, W, h)
    print "Total error: ", sum(abs(G1 - G2)), "h: ", h
    h /= 10
```

Figure 5: Test Case Used to Calculate Error Between Finite Difference Approximation and the Vectorized Gradient

Part 4

In this section, the neural network from part 2 was trained using vanilla gradient descent. The optimization procedure is described below, and learning curves and visualizations of the weights going into each of the output units are presented.

Optimization Procedure
Words

Learning Curves
Words

Weights Going Into Output Units
Words

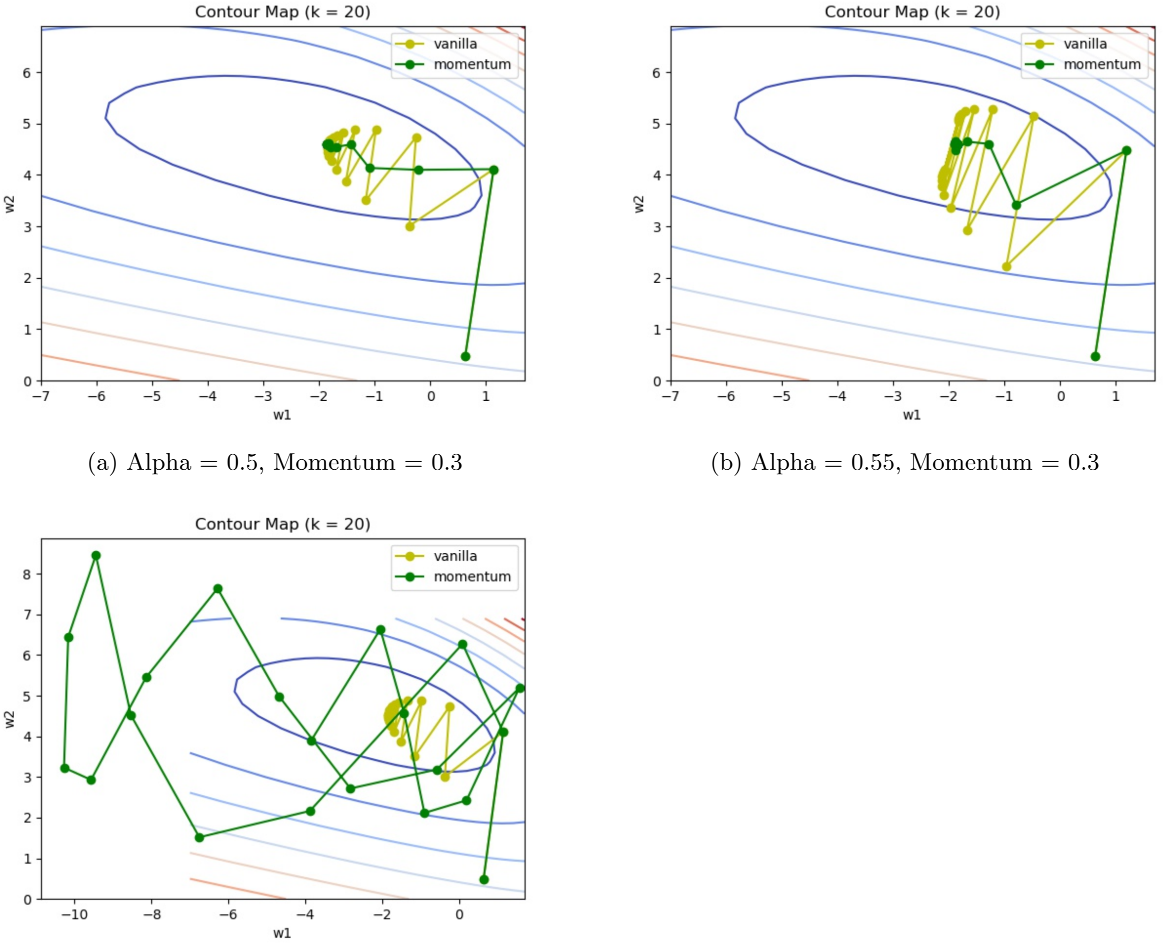
Part 5

Part 6

Comparison Between Momentum and Vanilla Gradient Descent
The effect of momentum on gradient descent was studied by taking two different weights, treating them as coordinates and plotting their path over 20 iterations on a contour plot. Gradient descent was run two times for each plot, using the same initial weight matrices and only changing the two chosen weights while keeping the other weights constant. The coordinates of the weights studied were $W[200, 2]$ and $W[201, 2]$. Figure 6a shows a plot where the two methods were comparable.

It was found that the effects of momentum could be seen most clearly when the learning rate was slightly increased, shown in Figure 6b. This is because in the vanilla method, the weights would travel mostly vertically, due to the shape of the contour, while in the momentum method, the vertical components of the weights would cancel each other out over consecutive steps.

However, when momentum is too high, the contributions from the previous steps would have too much of an effect, resulting in the weights overshooting the minimum like in Figure 6c.



(a) Alpha = 0.5, Momentum = 0.3

(b) Alpha = 0.55, Momentum = 0.3

(c) Alpha = 0.5, Momentum = 0.9

Part 7

Effect of Caching on the Time Required to Compute the Gradient
A neural network consists of several layers of *neurons* connected to each other by edges which each have their own *weight*. In order to optimize a neural network, we need to "learn" the values for the weights that minimize the cost function. The *gradient* of the cost function can be computed with respect to each weight in the network; this matrix-valued quantity indicates (up to a proportionality constant) how much each of the weights should be changed to achieve the target output. Traditionally, there are two algorithms that come to mind for how to compute the gradient, both of which make excessive use of the chain rule from calculus:

(1) Naïve approach: brute-force computation of the gradient, where each entry in the gradient matrix is evaluated separately.

(2) Better approach: "backpropagation", a computation of the gradient that uses caching to store each partial derivative so that no term ever has to be computed twice. Terms are accessed from a table in $O(1)$ time once they have been computed, and are added to the table otherwise.

This section presents an analysis of the runtime complexity of these two algorithms, ultimately deriving asymptotic upper bounds for each in the case of a fully-connected network with N layers and K neurons per layer. Note that only one training example will be considered, while in general the gradient would be an average over all training examples. The variables and dimensions to be used are explained as follows:

- C is the cost function
- $W^{(i,j,k)}$ is the weight corresponding to the edge connecting the j^{th} neuron in the $(i-1)^{\text{th}}$ layer to the i^{th} neuron in the i^{th} layer
  - $i \in [1, N]$
  - $j, k \in [1, k]$
- $a_k^{(i)}$ is the output from the k^{th} neuron in the i^{th} layer. Note that the neurons $a_k^{(0)}$ form the input layer and the neurons $a_k^{(N)}$ form the output layer.
- N is the number of layers in the network
- K is the number of neurons per layer

Also, the following reasonable assumptions are made to simplify the analysis:

1. Element-wise multiplication is $O(1)$
2. Addition is negligible
3. Assume that an N-layer neural network includes the input and output layers in the count; that is, it has $N-2$ hidden layers
4. Assume activations functions and non-linearities such as softmax at the output layer can be ignored
5. Partial derivatives in tables (i.e. previously-solved subproblems) take negligible time to retrieve

Characterization of Naïve Approach (Brute Force)
We begin with the naïve approach as this will make the analysis for the "better" approach more simple due to the terminology developed. We begin by writing the derivative of the cost function with respect to an arbitrary weight:

$$\frac{\partial C}{\partial W^{(i,j,k)}}$$

Next, we consider the fact that the cost function is evaluated at the output neurons, $a_k^{(N)}$; that is, $C = C(a_1^{(N)}, ..., a_K^{(N)})$. Each $a_k^{(N)}$ is in turn computed by first evaluating the inner product $\sum_{j=1}^{K} W^{(N-1,j,k)} a_j^{(N-1)}$, which uses the outputs from layer $N-1$ and the weights connecting these outputs to neuron k in layer N. Each $a_j^{(N-1)}$ can also be computed in a similar fashion using the outputs and weights from the previous layer, until layer 1 is reached, which is the input layer. What this means is that any $a_k^{(i)}$ is technically an input to the cost function, so it makes sense to write expressions such as $\frac{\partial C}{\partial a_k^{(i)}}$. In fact, we can relate this to the partial derivative with respect to an arbitrary weight in the network using the chain rule from calculus:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}}$$

But we can expand $\frac{\partial C}{\partial a_k^{(i)}}$ in terms of the a_k's that are affected in the next layer by an infinitesimal change in $a_k^{(i)}$ (note that we sum from 1 to K because we are studying a fully-connected network and hence $a_k^{(i)}$ connects to K neurons in the next layer):

$$\frac{\partial C}{\partial a_k^{(i)}} = \sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big)$$

This expansion gives yet a larger expression for the derivative of the cost function with respect to an arbitrary weight:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(i)}} \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} = \Bigg(\sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big) \Bigg) \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} = \frac{\partial a_k^{(i)}}{\partial W^{(i,j,k)}} \sum_{l=1}^{K} \Big(\frac{\partial C}{\partial a_l^{(i+1)}} \frac{\partial a_l^{(i+1)}}{\partial a_k^{(i)}} \Big)$$

Consider the case where we are interested in $\frac{\partial C}{\partial W^{(1,j,k)}}$, that is, the derivative of the cost function with respect to a weight connecting the input layer and layer 1. In this case, the expansion has N summation terms involving weights in the next layers:

$$\frac{\partial C}{\partial W^{(1,j,k)}} = \frac{\partial a_k^{(1)}}{\partial W^{(1,j,k)}} \sum_{l_1=1}^{K} \Big(\frac{\partial a_{l_1}^{(2)}}{\partial a_k^{(1)}} \sum_{l_2=1}^{K} \Big(\frac{\partial a_{l_2}^{(3)}}{\partial a_{l_1}^{(2)}} \sum_{l_3=1}^{K} \Big(... \qquad ... \sum_{l_{N-1}=1}^{K} \Big(\frac{\partial a_{l_{N-1}}^{(N-1)}}{\partial a_{l_{N-2}}^{(N-2)}} \sum_{l_N=1}^{K} \Big(\frac{\partial a_{l_N}^{(N)}}{\partial a_{l_{N-1}}^{(N-1)}} \Big)\Big)\Big)\Big)\Big)$$

Since there are N sums, and each runs from 1 to K, this computation of the gradient with respect to a weight between the input layer and the first layer involves k^N operations, provided that the previously-stated assumptions hold. Thus, it follows that for a weight connecting the first and second layers, there are k^{N-1} computations (since the indices in the previous expression would all begin at layer 2 instead of layer 1). Indeed, for the gradient with respect to any weight connecting neurons in the $(i-1)^{\text{th}}$ and i^{th} layers, there are $k^{N-(i-1)}$ computations. Thus, for weights connecting the $N-1^{\text{th}}$ and N^{th} layers, there are then $k^{N-(N-1)} = k$ computations.

The total number of operations is then:

$$k(k^N + K^{N-1} + ... + k) = k^{N+1} + k^N + ... + k^2 \in O(Nk^{N+1})$$

So the naïve algorithm has a runtime complexity that is asymptotically bounded above by $O(Nk^{N+1})$.

Characterization of Backpropagation

Now we will analyse the "better" solution, backpropagation. Intuitively, we expect that after computing each partial derivative once, we should never need to compute it again as this exemplifies an *overlapping subproblem*. To illustrate this, note that the partial derivative of the cost function in the first layer (implicitly) uses the partial derivatives of the cost function with respect to the outputs and weights in the second layer, and so forth. These latter partial derivatives should be retrievable in constant time if they were computed earlier.

Note the following:

1. Fully connected implies $k \times k = k^2$ weights per inter-layer junction
2. N layers implies $N-1$ inter-layer junctions
3. Each partial derivative should be constant time to compute
4. $(N-1)k^2$ total weights in the network

Now, in the final layer of the network, there is 1 partial derivative to compute for each $a_k^{(N)}$, which is simply $\frac{\partial C}{\partial a_k^{(N)}}$. However, in the previous inter-layer junction (i.e. connecting the $(N-1)^{\text{th}}$ layer to the N^{th} layer), there are 2 partial derivatives to compute, as shown below:

$$\frac{\partial C}{\partial W^{(i,j,k)}} = \frac{\partial C}{\partial a_k^{(N)}} \frac{\partial a_k^{(N)}}{\partial W^{(N,j,k)}}$$

But $\partial a_k^{(N)}$ is computed when considering the derivative of the cost function with respect to the output layer. So, if we work backwards through the neural network, we can see we can save time on computation since we don't need to keep computing partial derivatives whose values we already know. That is, in the case shown above, there were two partial derivatives that needed to be computed, but only one of them would need to be computed if the derivatives in the previous layer were already solved.

This concept generalizes across the network; thus, since there is one new computation required per weight, and there are k^2 weights per inter-layer junction, the total number of computations required to solve the gradient of the network with respect to all weights is:

$$k^2 + k^2 + ... + k^2 \qquad N \text{ times}$$

This gives an asymptotic upper bound on the runtime complexity of $O(k^2 N)$.

Part 8

Implementation of a Fully Connected Neural Network in PyTorch A fully connected neural network was implemented in PyTorch with a single hidden layer and 6 output nodes. The network was trained on the faces of the 6 actors from project 1: Lorraine Bracco, Peri Gilpin, Angie Harmon, Alec Baldwin, Bill Hader and Steve Carell. The number of photos used per actor for each set was 70 for the training set and 20 for both validation and test sets. To obtain each set, the uncropped photos from project1 were processed using the $make3Sets()$ found in *image_processing.py*.

Part 9

Part 10