CSC411: Project #2

Due on Friday, March 10, 2017

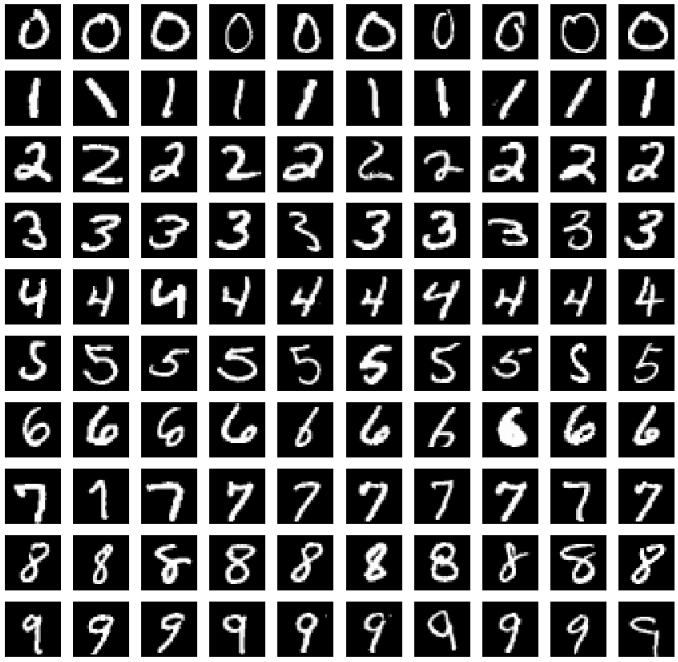
Fiona Leung *999766061*
Timur Borkhodoev *999376394*

March 11, 2017

Part 1

Dataset description The dataset contained an even number of images per digit, having no more or less information the neural networks coud train off of determine one digit better apart from another. Each digit had multiple images of it where it was drawn with varying writing styles. Some digits were written with more loops and imperfections than others, had different thicknesses or were drawn slanted at different angles.Some digit images had discontinuous lines in the number shapes. For example, in the images of the 0's some had a closed loop while others were perfect and had a small edge that jutted out near its top.

1. variety of angles
2. different styles of handwriting
3. gaps between continuous lines
4. different thickness levels



Part 2

Compute the network function by propagating forward and discarding intermediate results

```
def compute_network(x, W0, b0, W1, b1):
    _,_, output = forward(x, W0, b0, W1, b1)
    return = argmax(output)
```

Part 3

1. We will use negative log-probabilities as our cost function, and find its gradient

```
def cross_entropy(y, y_):
  return -sum(y_ * log(y))
```

2. Vectorized code for computing gradient of the cost function

```
def deriv_multilayer(W0, b0, W1, b1, x, L0, L1, y, y_):
    dCdL1 = y - y_
    dCdobydodh = dot(W1, dCdL1)
    dCdW1 = dot(L0, dCdL1.T)
    diff = 1 - L0**2

    dCdW0 = tile(dCdobydodh, 28 * 28).T * dot(x, (diff.T))
    dCdb0 = dCdobydodh * diff

    return dCdW1, dCdL1, dCdW0, dCdb0
```

To verify gradient correctness we used finite differences to compare our output.

Method	approximation	gradient
W1	0.984191894531	0.984342670068
W0	0.0	-0.0
b1	0.000476837158203	0.000534144113772
b0	0.0	0.0

Part 4

We start by loading sample weights from the pickle file. Then train it using mini-batch optimization to speed up training. Our training method works as follow:

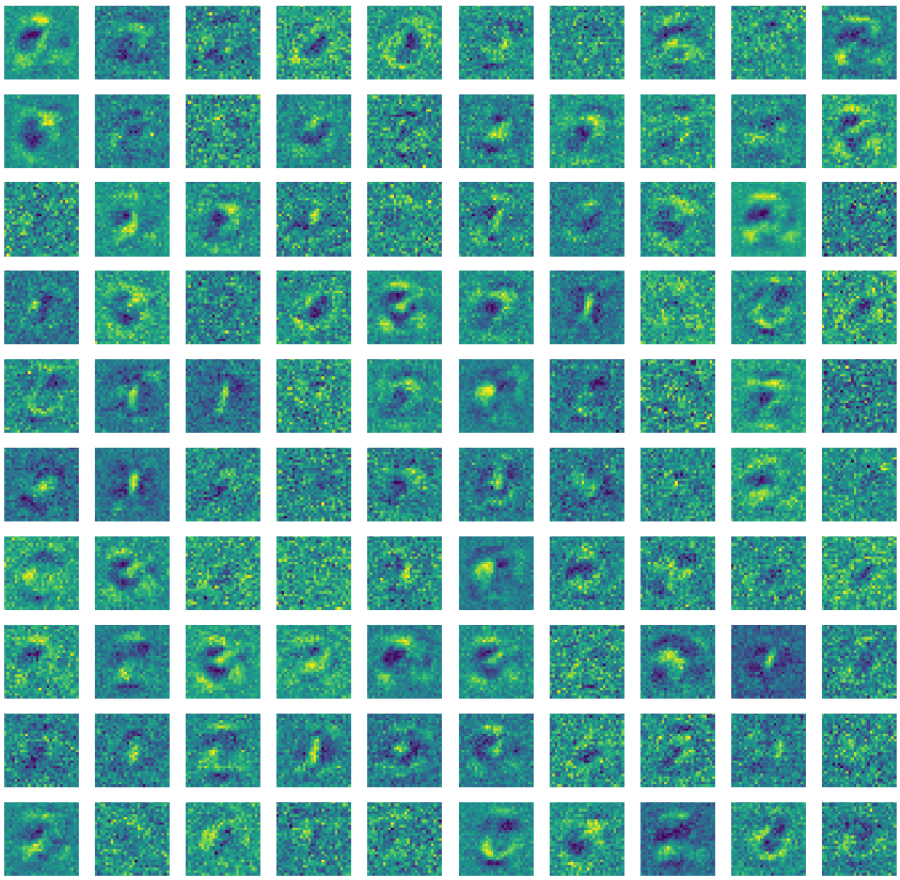
1. First forward method - passes flattened image through the network keeping all intermediate results
2. Then we compute the gradient with respect to $W0, b0, W1, b1$
3. We keep accumulating error values and then update our parameters $W0, b0, W1, b1$ over a single batch and repeat

```
def train(plot=False):
    global W0, b0, W1, b1
    global plot_iters, plot_performance
    plot_iters = []
    plot_performance = []
    alpha = 1e-3
    for i in range(150):
```

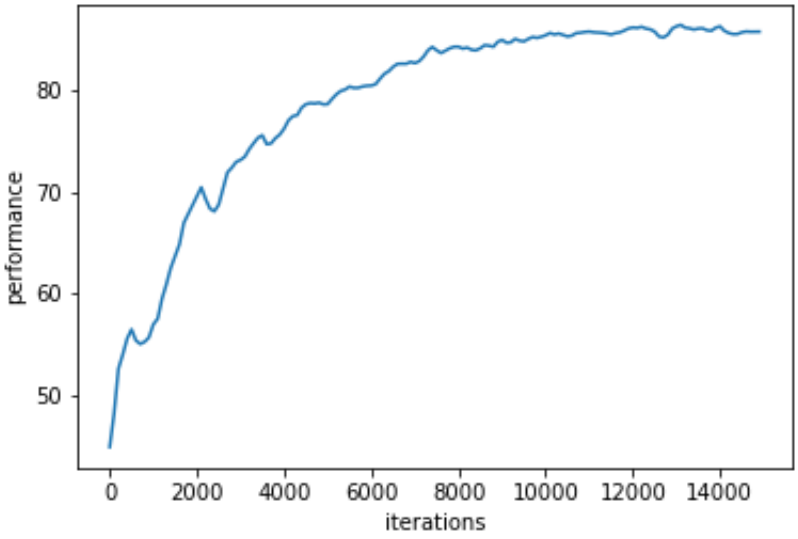
```
        X, Y, examples_n = get_batch(i * 5,10)

        update = np.zeros(4)

        for j in range(examples_n):
            y = Y[j].reshape((10, 1))
            x = X[j].reshape((28 * 28, 1)) / 255.
            L0, L1, output = forward(x, W0, b0, W1, b1)
            gradients = deriv_multilayer(W0, b0, W1, b1, x, L0, L1, output, y)
            update = [update[k] + gradients[k] for k in range(len(gradients))]

        # update the weights
        W1 -= alpha * update[0]
        b1 -= alpha * update[1]
        W0 -= alpha * update[2]
        b0 -= alpha * update[3]
        if plot:
            plot_iters.append(i * examples_n)
            plot_performance.append(test_perf())
    return plot_iters,plot_performance
train(plot=True)
```

```
def get_batch(offset,example_per_class=5):
    # 5 examples per class
    classes_num = 10
    x_batch = np.zeros((example_per_class * classes_num, 28 * 28))
    y_batch = np.zeros((example_per_class * classes_num, classes_num))
    for i in range(classes_num):
        for j in range(example_per_class):
            x_batch[i * example_per_class + j] = M['train' + str(i)][j +
                                                                     offset]
            y_batch[i * example_per_class + j][i] = 1
    return x_batch, y_batch, example_per_class * classes_num
```

visualizing weights:



performance graph:

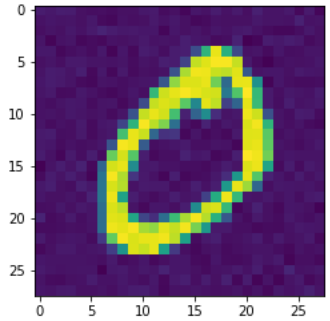


Part 5

So as discussed in lecture large errors are penalized quadratically in linear regressions. So our multinomial regression doesn't suffer from it. We start with generating a noise for our data set.

```
# generate noise and $N(0, \sigma^2)$
noise = scipy.stats.norm.rvs(scale=5,size=784*50*10)
noise = noise.reshape(500,784)
X,Y,n = get_batch(offset=0,examples=50)
X += noise
```

After modifying our data set with noise - we get images that look like this



The performance difference is drastic

Linear	Multinomial
43.54	82.92

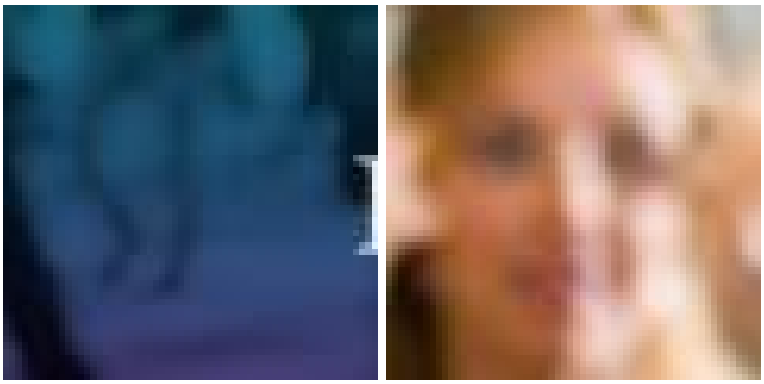
Part 6

Part 7

To construct the neural network, we trained on 5400 images with 90 images per actor. Each image was cropped to a bounding box that was resized to a 60×60 array containing only the actor's face. Images were serialized in a dictionary that mapped the actors' names to an array list of tuples containing the filename and RGB NumPy image arrays.
The dictionary's format looks like such:

```
actor_imgs = {
  'Steve Carell':
    [('carell1.jpg', carell1), ..., ('carelln.jpg', carelln)] carelln],
                                                    ...
  'Fran Drescher':
    [('drescher1.jpg', drescher1), ..., ('dreschern.jpg', dreschern)]
}
```

This dictionary was read from a file called `actor_imgs.p` using the `pickle` module. To create a unique training, validation or test set, we shuffled the images to select for these sets each time. By not hardcoding what images to use in the image set and shuffling, we could preventing the NN from detecting the images that appeared the most in these sets, removing biases.
Images were downloaded and serialized with the `get_all_imgs.py` script. **Note** that the images are RGB NumPy arrays resized to $60 \times 60 \times 3$ matrices then serialized here to make `faces.py` run faster.
The single `cPickle` file created is under `img_datatfactor_imgs.p`.
`faces.py` will open `actor_imgs.p` and filter out a hardcoded list of unusable images that are not of an actor's face or too blurry.



(a) Alec Baldwin (b) Chenoweth

Input Preprocessing
Each image passed through the neural network was preprocessed to standardize its values and characteristics so we could compare images on the same physical conditions as closely as possible.
To preprocess each image we did the following:

- grayscale the image to convert the 3D RGB image to a 2D array

- flattening the image array to a 1×3600 vector stored in a NumPy array
- normalize the image's pixels, their values are limited to values 0 to 1.

Weight Initialization
The weights were initialized to some random x and y value on a normal distribution with a standard deviation of 0.1. The function `tf.random.normal()` was used to generate these values.

Activation Function
Tanh activations were used in the hidden layer. The outputs of the tanh activations have a smaller chance of becoming a dead neuron during gradient descent.

Results
The neural network(NN) completed with the following results:

```
itr = 5000
Accuracy on:
    Training set 0.922222
    Validation set: 0.822222
    Test set: 0.85
```

The more iterations of gradient descent there were, the better the classfier's performance was on detecting the correct actor in an image since it could readjust it's weights to a much more accurate value closest to reproducing a clearer image of what each actor would look like.
The training set of course, would perform best across the other 3 image sets since the NN was trained on that image set unlike the validation and training sets.fig:performanceNN

Figure 2: NN Network Performance

Part 8

blah

Part 9

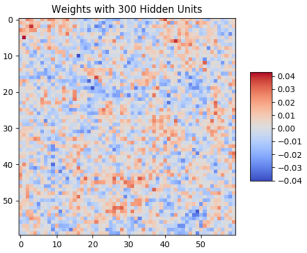
To generate the two sets of weights `W0 and W1`, we trained the NN on the same image set used in part 7 to adjust the weights and then return them in `part9()` of `faces.py`.
We created the function `visualize_weights()` to take in these weights, reshape them into the 60×60 pixel image, the size of the images the NN was trained on,, then displayed them. These results are automatically saved to image files named `images/part9_*.jpg`

Hidden Unit Selection To pick the best number of hidden units, we selected one based on its performance. We tested the NN on 300, 600 and 900 units, visualizing the weights at each step. **Results**

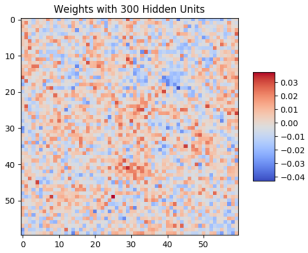
on 300 hidden units

```
itr = 5000
        Accuracy on:
            Training set 1.0
            Validation set: 0.85
            Test set: 0.838889
        On 600 hidden units --
```

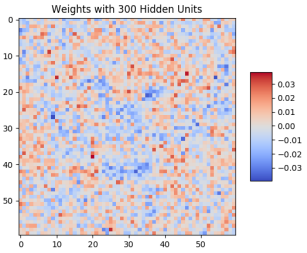
Weight Visualization



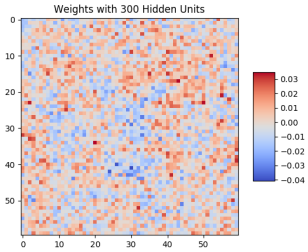
(a) Visualize weights with 50 units out of 300 hidden units



(b) Visualize weights with 100 units out of 300 hidden units



(c) Visualize weights with 200 units out of 300 hidden units



(d) Visualize weights with 300 units out of 300 hidden units

On 600 hidden units

```
itr = 5000
Accuracy on:
    Training set 1.0
    Validation set: 0.838889
    Test set: 0.833333
```

Code

Part 10

We extracted the conv4 activations after we trained AlexNet on our set of of actor images in `get_conv_activations()`, using 5400 images(150 images per actor) deserialized from a pickle file `/img_data/alexnet/actor_imgs_with_filename`
Then we fed these activations into our NN in `part10()` which would train over 500 iterations then test its performance accuracy at detecting the correct actor on the training, validation and training set. It's performance at each iteration is outputted when this function is run.
The NN performed signficantly better at classifying actors on the training set it was trained on versus the validation and training set.

```
itr = 500
        Accuracy on:
            Training set 0.977778
            Validation set: 0.772222
            Test set: 0.766667
```

System architecture
Activation Extraction
We extracted the conv4 activations that were made in AlexNet after we trained it on a set of $227 \times 227 \times 3$ RGB images of the actors in the function `get_conv4_activations()` from `deepfaces.py`. The images were deserialized from a pickled image file described below.
Training
We used about 150 images per actor, so 5400 images in total to train AlexNet. The images were gathered from `get_all_imgs.py`, this time setting `IMG_SHAPE` to `( 227, 227, 3 )` to match the image shapes already coded in the given AlexNet code from http://www.cs.toronto.edu/~guerzhoy/tf_alexnet/myalexnet_forward_newtf.py.
The image pickle file was created in the same format desribed in **Part 7** under `img_data/alexnet/actor_imgs_with_file` using the `get_all_imgs.py` script.
When this pickle file was deserialized it was preprocessed to remove any images that did not contain a distinguishable face, and was a 3D RGB image. The images selected in the training set were shuffled each time to randomize the training set and eliminate bias for faces that could reappear in the image set a fixed number of times.
Weight initialization
Weights were initialized using the weights file given on the class site, `bvlc_alexnet.npy`
Note that this file was too large to upload and include in our submission. It must be downloaded and put in the same root folder as `deepfaces.py` to run it.
`bvlc_alexnet.npy` can be downloaded from here: http://www.cs.toronto.edu/ guerzhoy/tf_alexnet/bvlc_alexnet.npy
Then, we fed the activations from `get_conv4_activations()` into `part10.py`