Week 7 Lab

The questions below are due on Sunday November 05, 2017; 11:00:00 PM.

You are not logged in.

If you are a current student, please Log In (https://introml.mit.edu/fall17/labs/lab07?loginaction=login) for full access to this page.

A code and data folder that will be useful for doing this lab can be found here (https://introml.mit.edu/__STATIC__/fall17/labs/lab07/code_for_lab7.py.zip). Download this to your computer.

This lab is divided into two parts:

1. Checkoff on Neural Networks
2. Implementing Neural Networks

Part 1 will require a checkoff with a staff member.

A code and data folder that will be useful for doing this lab can be found here (https://introml.mit.edu/__STATIC__/fall17/labs/lab07/code_for_lab7.py.zip). Download this to your computer.

1) NEURAL NETWORKS

1.1) Crime and Punishment

One important part of designing a neural-network application is understanding the problem domain and choosing

- Representation for the input
- The number of output units and what values they can take on
- The loss function that we will try to minimize, based on actual and desired outputs

We have studied input representation in a previous lab, so in this lab we will concentrate on the number of output units, activation function on the output units, and loss function. These should generally be chosen jointly.

Just as a reminder, we have studied:

- Activation functions: linear, relu, sigmoid, softmax
- Loss functions: hinge, NLL (also known as cross-entropy), quadratic

For each of the following application domains, specify the number of units in the output layer, the activation

function(s) on the output layer, and the loss function. When you choose to use multiple output units, be very clear on the details of how you are applying the activation and the loss. Write your answers down!

1. Map the words on the front page of the New York Times to the predicted (numerical) change in the stock market average
2. Map a satellite image centered on a particular location to a value that can be interpreted as the probability it will rain at that location some time in the next 4 hours
3. Map the words in an email message to which one of a user's fixed set of email folders it should be filed in
4. Map the words of a document into a vector of outputs, each of which represents a topic, and has value 1 if the document addresses that topic and 0 otherwise. Each document may contain multiple topics, so in the training data, the output vectors may have multiple 1 values.

1.2) Architecture

We experimented with three different architecture/training combinations:

- m hidden units, no regularization
- $2m$ hidden units, no regularization
- $2m$ hidden units, dropout regularization

We observed that

- Model A: training error 0.2, validation error 0.35
- Model B: training error 0.25, validation error 0.3
- Model C: training error 0.1, validation error 0.4

Please assign the models A,B, and C to their best fitting setups above.

1.3) Checkoff

Checkoff 1:

Have a check-off conversation with a staff member, to explain your answers.

Ask for Help | Ask for Checkoff

2) IMPLEMENTING NEURAL NETWORKS

Although for "real" applications you want to use one of the many packaged implementations of neural networks (we'll start using one of those next week), there is no substitute for implementing one yourself to get an in-depth understanding. Luckily that is relatively easy to do if we're not too concerned with maximum efficiency.

We will specify a network with L layers as follows:

- `sizes` a list of L layer sizes, starting with the size of the input layer
- `weights` a list of $L-1$ $n \times m$ matrices where m is the size of layer $l-1$ and n is the size of layer l.
- `biases` a list of $L-1$ column vectors of size $n \times 1$, where n is the size of layer l, one for each layer beyond the input.
- `act_funs` a list of $L-1$ activation functions, one for each layer beyond the input. Each function takes the layer's weighted input (z^l) and returns the activation ($a^l = f(z^l)$).
- `act_deriv_funs` a list of $L-2$ derivatives for the layer activation functions, one for each layer beyond the input but not including the final layer. Each function takes the layer's weighted input (z^l) and returns $\frac{\partial a^l}{\partial z^l}$. The derivative for the output layer activation is handled by the `loss_delta_fun`.
- `loss_fun` a loss function for the network, which takes the final layer activations and the target and return $C(a^L, y)$.
- `loss_delta_fun` a derivative function for the loss, which takes the final layer weighted inputs, activations and the target and return δ^L (which depends on the gradient of the loss and the derivative of the activations).
- `class_fun` given the last activation layer returns a class index, useful for classification.

Our implementation will have a class structure as follows; you can find this in the file provided with the lab. That will also contain an example network and test data.

```
class NN:
    sizes = []
    act_funs = []
    act_deriv_funs = []
    loss_fun = None
    loss_delta_fun = None
    class_fun = None
    def initialize(self):
        # sets self.weights and self.biases
        pass
    def forward(self, x):
        pass
    def backward(self, x, y):
        pass
    def sgd_train(self, X, Y, n_iter, step_size):
        pass
    def evaluate(self, X, Y):
        count = 0                       # number of errors
        loss = 0                        # cumulative loss
        d, n = X.shape                  # d = # features, n = # points
        o, _ = Y.shape                  # o = # outputs, n = # points
        for i in range(n):
            zs, activations = self.forward(X[:,i:i+1]) # compute activations
            act_L = activations[-1]                    # last layer
            pj = self.class_fun(act_L)                 # predict a class
            y = Y[:,i:i+1]                             # ith target
            yj = self.class_fun(y)                     # predict target
            if pj != yj:
                count += 1              # increment # wrong
            loss += self.loss_fun(act_L, y) # increment loss
        # pct error, average loss
        return count/float(n), loss/float(n)
```

The function `make_nn(D, hidden, O, init=True)` is used in most of the tests in this lab. `D` is the size of the input layer, `hidden` is a list of sizes of the hidden layers and `O` is the size of the output layer.

To simplify checking, we'll ask you to write each method independently.

2.1) Initialize

Our first step is to initialize the weights and biases. We will test that the size of the arrays created are the correct sizes. The values for the weights should be small random numbers with a distribution as specified in the notes; the biases can start out as zero.

You should be aware of the numpy function `np.random.normal` for generating normally distributed random variables.

```
def initialize(self):
    # initialize the biases to zero, no bias on input layer
    # initialize the weights to normal with variance 1/m (deviati
    # return network to enable checking
    sigma = 1/np.sqrt(self.sizes[0]) #sizes[0] - units on the inp
    mu = 0
    self.weights = [np.random.normal(mu, sigma, shape) for shape
                    zip(self.sizes[1:], self.sizes[:-1])]
    self.biases = [np.zeros((i,1)) for i in self.sizes[1:]] #bias
    return self
```

Ask for Help

2.2) Forward

Now we need to write the `forward` method that takes an input vector and returns a list of the weighted inputs and the activations for each layer in the network. Specifically:

- `x` is a column vector of inputs, the activations of the input layer

It returns a tuple of two lists:

- `zs` a list of column vectors of weighted inputs for each layer beyond the input
- `activations` a list of column vectors of activations for each layer, including the input as the first entry

Note that all the multiplications of the weights times the inputs as a single matrix multiply per layer!

To help you debug, see the example (https://introml.mit.edu/__STATIC__/fall17/labs/lab07/nn_example.html). These are sample network with the same shape as the test networks and showing the sequence of computations during the forward and backward passes.

```
def forward(self, x):
    # Your code here, return (zs, activations)
    sigma = 1/np.sqrt(self.sizes[0]) #sizes[0] - units on the inpu
    mu = 0
    self.weights = [np.random.normal(mu, sigma, shape) for shape i
                    zip(self.sizes[1:], self.sizes[:-1])]
    self.biases = [np.zeros((i,1)) for i in self.sizes[1:]] #biase
    return self

```

Ask for Help

2.3) Backward

Now we need to write the `backward` method that takes an input vector (x) and a target activation (y) and returns the gradient of the loss wrt weights and the gradient of the loss wrt biases. We will use this function to compute the gradient for a step of SGD.

Specifically:

- `x` is a column vector of inputs, the activations of the input layer
- `y` is a column vector of targets, the desired activations of the output layer

It returns a tuple of two matrices:

- `grad_w` is a list of matrices, each is the gradient of the loss wrt weights for a layer, for the input point
- `grad_b` is a list of vectors, each is the gradient of the loss wrt biases for a layer, for the input point

Recall the four basic equations for backprop in Homework 7, Problem 5.

$$(A) \quad \delta_j^L = \frac{\partial C}{\partial a_j^L} f'(z_j^L), \qquad (B) \quad \delta_j^l = \sum_k w_{kj}^{l+1} \delta_k^{l+1} f'(z_j^l).$$

$$(C) \quad \frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l, \qquad (D) \quad \frac{\partial C}{\partial b_j^l} = \delta_j^l.$$

We can write matrix/vector forms of these equations, which lead a compact form of the backprop algorithm:

- Recall also that part of our specification for the network is a function `loss_delta_fun(z, a, y)` that returns the vector δ^L as in equation (A).
- Equation (B) is defines how the errors propagate backwards

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) * f'(z^l)$$

where the first term is a matrix/vector producy and $*$ indicates component-wise multiply.

- Equation (C) is the gradient wrt weights

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

where the operation on the right is a matrix multiply (an outer product).

- Equation (D) is the gradient wrt biases

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

To help you debug, see the example (https://introml.mit.edu/__STATIC__/fall17/labs/lab07/nn_example.html). These are sample network with the same shape as the test networks and showing the sequence of computations during the forward and backward passes.

```
def backward(self, x, y):
    # Your code here, return (grad_w, grad_b)
    zs, actvns = self.forward(x)
    grad_bs = [self.loss_delta_fun(zs[-1], actvns[-1], y)]
    grad_ws = [np.dot(grad_bs[0], actvns[-2].T)]

    for i in range(len(self.sizes)- 3, -1, -1): #gradient for the
        delta_l = np.dot(self.weights[i+1].T, grad_bs[0]) \
                  * self.act_deriv_funs[i](zs[:-1][i])
        grad_w = np.dot(delta_l, actvns[i].T)
        grad_bs.insert(0,delta_l)
        grad_ws.insert(0, grad_w)
    return (grad_ws, grad_bs)

```

Ask for Help

2.4) SGD_Train

Now we need to write the `sgd_train` method. This should be a (by now familiar) application of the stochastic gradient descent approach, using the backward function to compute the gradient. The method takes a dataset as input and specification of the step size and number of iterations.

Specifically:

- `X` is a dxn data matrix
- `Y` is a oxn matrix of targets, where o is the size of the output activation layer

It should return the network for checking,

You should use `np.random.randint` to choose the random index into the data for SGD

```
 6     min_error = 1
 7     while count < n_iter:
 8         idx = np.random.randint(X.shape[1]-1)
 9         grad_w, grad_b = self.backward(X[:,[idx]], Y[:,[idx]])
10         self.weights = [w for w in map(lambda w, dw: w - (step_
11                                        self.weights, grad_w)]
12         self.biases = [b for b in map(lambda b, db: b - (step_s
13                                       self.biases, grad_b)]
14         count += 1
15         error_rate = self.evaluate(X, Y)
16         print ('Current Error Rate:', self.evaluate(X, Y))
17         if error_rate[0] < min_error:
18             min_error = error_rate[0]
19         if error_rate[0] <= 0.005:
20             break
21     print ('DONE TRAINING. Iterations taken:', count)
22     print ('Final Error Rate:', self.evaluate(X, Y))
23     print ('Min error_rate:', min_error)
24     return self
25
```

Ask for Help

2.5) Making a network

We now have all we need to train a network, except for the components of the network spec, the activations, loss, etc. We will now make a network suitable for classification tasks. The following function makes such a network. You should make sure that you understand all the components

```
def classify(X, Y, hidden=[10, 10], it=10000, lr=0.005):
    D = X.shape[0]
    N = X.shape[1]
    O = Y.shape[0]
    # Create the network
    nn = NN()
    nn.sizes = [D] + list(hidden) + [O]
    nn.act_funs = [relu for l in list(hidden)] + [softmax]
    nn.act_deriv_funs = [relu_deriv for l in list(hidden)]
    nn.loss_fun = nll
    nn.loss_delta_fun = nll_delta
    nn.class_fun = softmax_class     # index of class
    # Modifies the weights and biases
    nn.sgd_train(X, Y, it, lr)
    return nn
```

To actually use this we need to define the functions relevant to activations, loss and making a class prediction:

- `relu(z)` is the ReLU activation for the hidden units ($f(z)$)
- `relu_deriv(z)` computes the derivative of the relu activations wrt z ($f'(z)$)
- `softmax(z)` activation for the last layer
- `softmax_class(a)` return a 0-based index for the class from the activations
- `nll(a, y)` negative log likelihood loss
- `nll_delta(z, a, y)` the value of δ^L, this has a particularly simple form for NLL loss and softmax output activation.

You should be aware of the numpy functions: `np.maximum`, `np.where`, `np.exp`, `np.sum` and `np.argmax`. Using these numpy functions, the required functions are all one liners.

```
 1 def relu(z):
 2     return np.maximum(z, np.zeros(z.shape))
 3 def relu_deriv(z):
 4     return np.where(z > np.zeros(z.shape), np.ones(z.shape), np.z
 5 def softmax(z):
 6     return  np.exp(z)/np.sum(np.exp(z))
 7 def softmax_class(a):
 8     return np.argmax(a)
 9 def nll(a, y):
10     return -np.sum(y*np.log(a))
11 def nll_delta(z, a, y):
12     # This is a very cool result, the f' term cancels out for sof
13     return a - y
14
```

Ask for Help

3) EXPERIMENT

That's it! A fully functional neural network training program.

You can see that the evaluate method makes predictions with trained network and prints the error rate (in percent points misclassified) and the value of the loss. During training (iterations of `sgd_train`), we want to print out these numbers on the training set and, preferably, also on a "validation set" - a data set separate from the training set.

You should copy your answers (or ours) into the file provded with the lab. There you can run the network on the couple of data sets provided there. Notably the dataset defined by `hard()` is the classification dataset that we used in Lab 6, Problem 1.5.1. Go back to that problem and run t1 with order = 3, which perfectly separates the data, keep that window visible.

Try to get a neural network to reach zero error rate on that same data:

```
X, Y = hard()
classify(X, Y, ...)
```

Try it with one hidden layer of size 10, then one of size 50. How many iterations were needed for each?

Perfectly separating a data set like this illustrates the power of the neural net to discover its own features, note that we did not have to specify features or a kernel. But, it doesn't mean that in practice we want to actually train a network until it fully separates the training data; we want to use validation data to choose how much training to do. Typically we want to stop training when the "validation error" stops dropping.