CS189: HW6

Lucine Oganesian
SID:23391268

April 27, 2016

Neural Network: Back-propagation Derivation

Before implementing the neural network it was necessary to calculate the gradient for two loss functions: mean square and cross-entropy.

Mean Square

Can rewrite the mean square cost function in matrix notation as

$$\frac{1}{2}\|Y - Z(x)\|_2^2 = \frac{1}{2}(Y^T Y - 2Y^2 Z(x) + Z'(x)Z(x)) = J$$

In order to perform gradient descent we need to calculate $\bigtriangledown J_W$ and $\bigtriangledown J_V$. Starting with $\bigtriangledown J_W$, we take the derivative of the above expanded expression and we get that

$$\bigtriangledown J_W = \frac{1}{2}(-2Y. * \frac{\partial Z}{\partial W} + 2Z(x). * \frac{\partial Z}{\partial W}) = (Z(x) - Y). * \frac{\partial Z}{\partial W}$$

Next we compute $(Z(x) = sig(Wh))$:

$$\frac{\partial Z}{\partial W} = sig'(Wh). * \frac{\partial Wh}{\partial W}$$

From matrix calculus we know that the derivative of Wh with respect to W is a 10×201 (num output by num hidden units) matrix, where each row is h^T. I will call this matrix $\hat{h^T}$. Thus (note $Z(x) = sig(Wh)$):

$$\bigtriangledown J_W = (Z(x) - Y). * sig'(Wh). * \frac{\partial Wh}{\partial W} = (sig(Wh) - Y). * sig(Wh). * (1 - sig(Wh)). * \hat{h^T}$$

(using notation introduced previously). Alternatively we can write this row-wise, such that

$$\bigtriangledown J_{W_i} = (sig(W_i h) - Y_i) * sig(W_i h) * (1 - sig(W_i h)). * h$$

(Note that $.*$ denotes element-wise multiplication of the vectors.)
Next we need to calculate the gradient for $\bigtriangledown J_V$.
The first part is similar to before:

$$\bigtriangledown J_V = \frac{1}{2}(-2Y. * \frac{\partial Z}{\partial V} + 2Z(x). * \frac{\partial Z}{\partial V}) = (Z(x) - Y). * \frac{\partial Z}{\partial V}$$

Furthermore,

$$\frac{\partial Z}{\partial V} = sig'(Wh). * \frac{\partial Wh}{\partial V}$$

The second partial derivative is computed as such

$$\frac{\partial Wh}{\partial V} = W. * \frac{\partial h}{\partial V}$$

Next,

$$\frac{\partial h}{\partial V} = tanh'(VX). * \frac{\partial VX}{\partial V} = tanh'(VX). * \hat{X^T}$$

where similar to before $\hat{X^T}$ denotes a 200×11 matrix (num hidden units by num inputs), where each row is X^T. Thus overall, $\bigtriangledown J_V$ can be written in matrix notation as (where $.*$ denotes element-wise multiplication):

$$(((Z(x) - Y). * sig(Wh). * (1 - sig(Wh)))^T W)^T. * 1 - tanh^2(VX). * \hat{X^T}$$

This can more clearly be written in row-wise notation

$$\bigtriangledown J_{V_j} = [\sum_i (Z(x)_i - Y_i) * sig(W_i h) * (1 - sig(W_i h)) * W_{ij}] * (1 - tanh^2(V_j X)). * X$$

Cross-entropy

Calculating cross entropy will yield the same exact expressions as above, except we need to use the appropriate derivative of the cost function

$$J = -\sum_{k=1}^{n_{out}} [Y_k \ln Z_k(x) + (1 - Y_k) \ln(1 - Z_k(x))]$$

Taking the derivative of this cost function with respect to W_i, with respect to a row of W, is (assume we are only working with one sample):

$$(-\frac{Y_i}{Z_i(x)} + \frac{1 - Y_i}{1 - Z_i(x)}) * \frac{\partial Z}{\partial W_i}$$

We already know what $\frac{\partial Z}{\partial W_i}$ is from the previous gradient calculation. Thus, taking this derivative and swapping it with the previous cost function's derivative in the row-wise expression from above

$$\bigtriangledown J_{W_i} = (-\frac{Y_i}{Z_i(x)} + \frac{1 - Y_i}{1 - Z_i(x)}) * sig(W_i h) * (1 - sig(W_i h)). * h$$

where $.*$ refers to element-wise multiplication.
Similarly for taking the derivative with respect to V_j, with respect to a row of V:

$$\sum_i (-\frac{Y_i}{Z_i(x)} + \frac{1 - Y_i}{1 - Z_i(x)}) * \frac{\partial Z}{\partial V_j}$$

We know $\frac{\partial Z}{\partial V_j}$ from calculating the gradient earlier (for mean square cost function). thus substituting the new loss function in place of the old loss function, we get (in row-wise notation)

$$\bigtriangledown J_{V_j} = [\sum_i (-\frac{Y_i}{Z_i(x)} + \frac{1 - Y_i}{1 - Z_i(x)}) * sig(W_i h) * (1 - sig(W_i h)) * W_{ij}] * (1 - tanh^2(V_j X)). * X$$

Now that the stochastic gradient update has been calculated a neural network can be implemented.

Implementing the neural network

See appendix for all python implementation.
A neural network with 10 input nodes, 1 hidden layer with 200 nodes, and an output layer with 10 nodes was implemented. A bias of 1 was added to both the input and hidden layer. Two cost functions were implemented, mean square error and cross-entropy, as defined in the homework specifications. All weights were randomly initialized from a Gaussian distribution. Momentum was implemented, learning rate decreased every epoch during training based on a function of the number of iterations, and all data was preprocessed. All networks were trained using stochastic gradient descent. See the implementation details below for specific parameter values.

Mean Squared Error Cost Implementation

I trained a neural network using a mean-squared error cost function. My **initial learning rate was set to 0.1 and decreased every epoch** as a function of the number of iterations

$$LR(t) = \frac{LR(0)}{1 + \frac{t}{\alpha |X|}}$$

Where t corresponds to the number of times the weights have been updated and $|X|$ corresponds to the number of samples we are training on. α was a parameter available for tuning. I ended up just **using** $\alpha = 1$ to make the learning rate reassignment related to length of the epoch (thus the learning rate changes according to the number of epochs that have happened, in addition to the number of times the weights have been updated). The following annealing method was based off of a method described in an online source [1]. If the learning rate was initialized any larger, then algorithm ended up getting stuck in a local minima and performining poorly. Slower learning rates normally resulted in a long training period.
The neural network **weights were initialized by random assignment from a normal distribution with zero mean and a standard deviation of 0.01**. This assignment was chosen based off of recommendations on piazza. I also implemented momentum as seen in class:

Algorithm 1 momentum

1: $\Delta w = -\epsilon \bigtriangledown w$
2: **while** training **do**
3: w = w + Δw
4: $\Delta w = -\epsilon \bigtriangledown w + \beta \Delta w$

I used a β **hyper-parameter value of 0.01**. Using higher values of beta (i.e, 0.1 and 0.5) often would make the training period take a long time, whereas using smaller β values (i.e, 0, or no momentum) would result in the algorithm getting stuck in local minima.
I computed training error and the total cost of the current set of weights every 1000 updates. The following figures represent the training error and total cost of the current set of weights respectively:
Final training accuracy was 2.1%, implying that the network was actually starting to overfit.
Before training the neural networks, I preprocessed the training data and normalized it to have a mean of 0 and a standard deviation of 0.5 (to speed up the training process). I normalized the validation data using the training data's mean and standard deviation, such that the validation data would be shifted the same way as the training data. (I empirically compared normalizing the validation data with its own mean and standard deviation against normalizing it with the training data's mean and standard deviation, the validation accuracy of the former was 5.04% error, whereas the latter had 4.2% error, suggesting marginal improvement of performance.)
Training took approximately 15 minutes (4 epochs). **Validation accuracy after full training as 4.2% error.** There were 50,000 training images and 10,000 validation images.
I determined when to stop training based off of the cost of the current set of weights, which was computer every 1000 updates and printed out. The amount of improvement in cost became non-significant after 4 epochs, which is when I chose to terminate the training (you can see in the figure plotting cost vs. training iteration that the curve effectively flat lines after a while and the change is minimal).

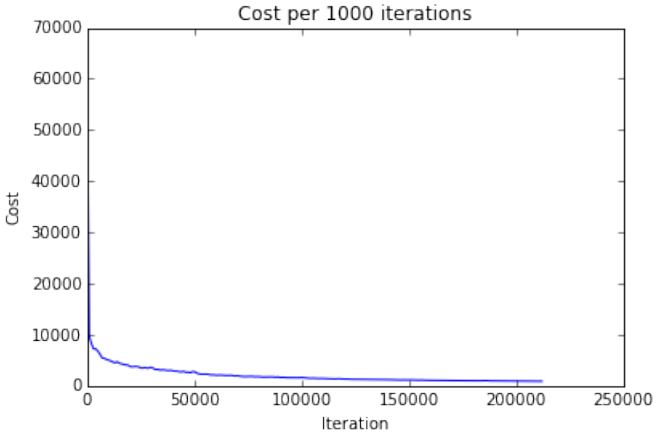

Figure 1: Iteration vs cost of current weights

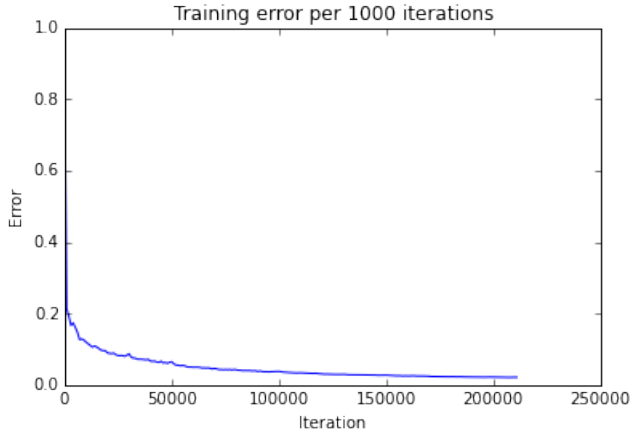

Figure 2: Iteration vs. training error

Cross entropy cost implementation

I trained a neural network using a cross-entropy error cost function. My **initial learning rate was set to 0.025 and decreased every epoch** as a function of the number of iterations (same as for mean-squared)

$$LR(t) = \frac{LR(0)}{1 + \frac{t}{\alpha |X|}}$$

Similar to mean-squared error, I set $\alpha = 1$. The initial learning rate for cross entropy significantly impacted the performance of the neural network. If the step size was initially too large, overall cost after each updated fluctuated too much and didn't decrease as much as it could (i.e. it was getting stuck in local minima and performing poorly).

The neural network **weights were initialized by random assignment from a normal distribution with zero mean and a standard deviation of 0.01**. This assignment was chosen based off of recommendations on piazza. I also implemented momentum and used a β **hyper-parameter value of 0.01**, similar to the mean-squared error case for similar reasons.

I computed training error and the total cost of the current set of weights every 1000 updates. The following figures represent the training error and total cost of the current set of weights respectively:

Final training accuracy was 1.5%, implying that the network was actually starting to overfit.

Before training the neural networks, I preprocessed the training data to be binarized: if a particular pixel value was non-zero, set the feature value to be 1, else set it to 0. I preprocessed all training and test data similarly.

Training took approximately 16 minutes (a little over 4 epochs). **Validation accuracy after full**

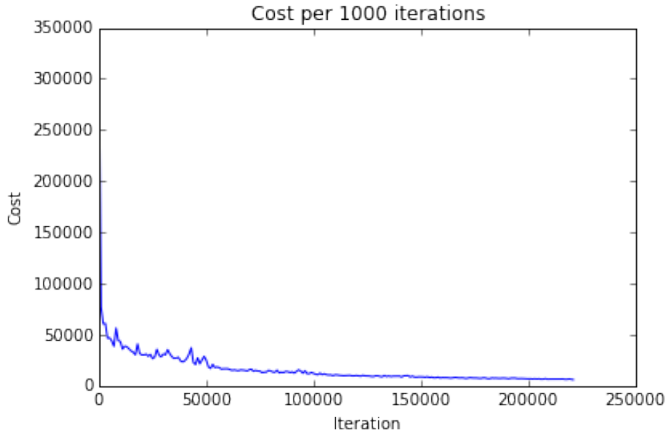

Figure 3: Iteration vs cost of current weights

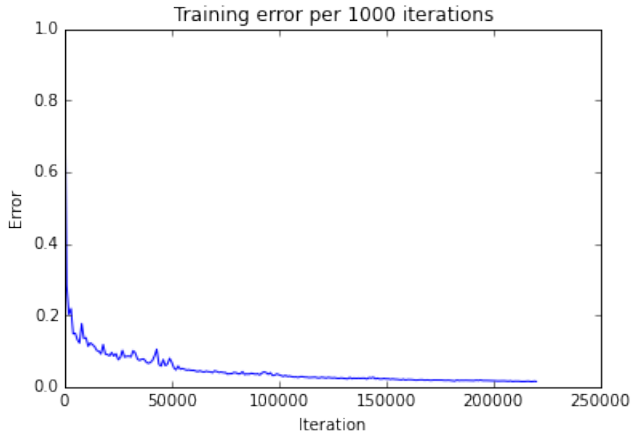

Figure 4: Iteration vs. training error

training as 3.81% error. There were 50,000 training images and 10,000 validation images.
I determined when to stop training based off of the cost of the current set of weights, which was computer every 1000 updates and printed out. The amount of improvement in cost became non-significant after approximately 4 epochs, which is when I chose to terminate the training (you can see in the figure plotting cost vs. training iteration that the curve effectively flat lines after a while and the change is minimal).

Which was better?

I spent more time tuning the parameter values for cross-entropy, it was initially underperforming mean-squared error. However, once I found the right set of parameters, cross-entropy performed marginally better, as it should because this is a classification problem, not a regression problem. When plotting the training error and the cost every 1000 iterations, it looked like cross entropy fluctuated a lot more than mean squared error, even though it also yielded a similar learning curve. Cross entropy took slightly less time to train as compared to mean squared error (when training on the full 60,000 images; they took about the same time for the 50,000 images when doing the validation testing).
My **best kaggle score** for using **mean-squared error** was **0.96400** using a training set of 60,000 images and the same parameters and preprocessing methods mentioned above in the mean-squared error section.
My **best kaggle score** for using **cross-entropy cost functions** was **0.96460**, only marginally better. The neural network was trained on 60,000 images using the same parameters as mentioned above under the cross entropy section. The training data was preprocessing using the same binarization method mentioned above.

Appendix

See all python implementation here.
The following is the neural network implementation.

```
class NeuralNetwork():
    def __init__(self, inputSize, outputSize, numHiddenUnits=200):
        self.numInput = inputSize
        self.numHiddenUnits = numHiddenUnits
        self.numOutput = outputSize

    def benchmarck(self, predY):
        return np.sum(np.argmax(self.Y, axis=0) != predY)

    def sigmoid(self, x):
        try:
            tmp = 1 / (1 + np.exp(-x))
        # addressing overflow errors
        except OverflowError:
            print 'overflow'
            if x > 0:
                tmp = 1
            elif x < 0:
                tmp = 0
        return tmp

    def sigmoidPrime(self, x):
        tmp = self.sigmoid(x) * (1 - self.sigmoid(x))
        return tmp

    def tanh(self, x):
        return np.tanh(x)

    def tanhPrime(self, x):
        return 1 - self.tanh(x)**2

    def meansquareCost(self, X=None, Y=None):
        # calculate cost for full X
        if X is None and Y is None:
            predY = self.forward()
            return 0.5 * np.sum(LA.norm(predY-self.Y, ord=2, axis=0)**2)
        # calculate cost for 1 sample
        else:
            predY = self.forward(X=X, train=False)
            return 0.5 * np.dot((predY-Y).T, predY-Y)

    def meansquaredPrime(self, X=None, Y=None):
        if X is None and Y is None:
            X, Y = self.X, self.Y
        predY = self.forward(X=X, train=False)
        return (predY-Y)

    def crentropyCost(self, X=None, Y=None):
        # calculate cost for full X
        if X is None and Y is None:
            predY = self.forward()
            predY = self.entropyOffset(predY)
            return - np.sum(np.sum(self.Y * np.log(predY) +
            (1 - self.Y) * np.log(1-predY), axis=0))
        # calculate cost for 1 sample
```

```
        else:
            predY = self.forward(X=X, train=False)
            predY = self.entropyOffset(predY)
            return - np.sum(Y * np.log(predY) + (1 - Y) * np.log(1-predY))

    def crentropyPrime(self, X=None, Y=None):
        if X is None and Y is None:
            X, Y = self.X, self.Y
        predY = self.forward(X=X, train=False)
        predY = self.entropyOffset(predY)
        return - (Y/predY) + (1-Y)/(1-predY)

    def entropyOffset(self, Y):
        Y[Y==0] += 0.0000000001
        Y[Y==1] -= 0.0000000001
        return Y

    def callCost(self, X=None, Y=None, costFunc='meansq', prime=False):
        if prime:
            if costFunc == 'meansq':
                cost = self.meansquaredPrime(X=X, Y=Y)
            elif costFunc == 'crentropy':
                cost = self.crentropyPrime(X=X, Y=Y)
        else:
            if costFunc == 'meansq':
                cost = self.meansquareCost(X=X, Y=Y)
            elif costFunc == 'crentropy':
                cost = self.crentropyCost(X=X, Y=Y)
        return cost

    def callActivation(self, X, activation, prime=False):
        if prime:
            if activation == 'tanh':
                out = self.tanhPrime(X)
            elif activation == 'sig':
                out = self.sigmoidPrime(X)
        else:
            if activation == 'tanh':
                out = self.tanh(X)
            elif activation == 'sig':
                out = self.sigmoid(X)
        return out

    def saveWeights(self, costFunc='meansq'):
        print 'Iteration {0}. Saving weights...'.format(self.numIter)
        pickle.dump((self.V, self.W),
        open('weights_{0}iter.pkl'.format(self.numIter), 'wb'), protocol=-1)

    def numericalGradient(self, samp, label, costFunc='meansq', eps=0.1):
        original = self.V[10,10]

        self.V[10,10] += eps
        leftCost = self.callCost(X=samp, Y=label, costFunc=costFunc)

        self.V[10,10] = original - eps
        rightCost = self.callCost(X=samp, Y=label, costFunc=costFunc)

        self.V[10, 10] = original
        return (leftCost-rightCost)/(2*eps)
```

```
def assignLearningRate(self, factor=1):
    self.learnRate = self.startLR/(1+self.numIter/(factor*self.X.shape[1]))

def forward(self, X=None, train=True, returnIntermediate=False):
    X = self.X if train else X

    VX = np.dot(self.V, X)
    h = self.callActivation(VX, self.hiddenact)
    if self.bias:
        h = np.vstack((h, np.ones((1, h.shape[1])))) # add bias if necessary
    Wh = np.dot(self.W, h)
    Z = self.callActivation(Wh, self.outact)

    # return intermediate products and activations as well
    if returnIntermediate:
        return VX, h, Wh, Z
    # return just final output
    return Z

def backwardProp(self, X=None, Y=None, cost='meansq'):
    if X is None and Y is None:
        X, Y = self.X, self.Y
        # self.X, self.Y = X, Y

    dL = self.callCost(X=X, Y=Y, costFunc=cost, prime=True)
    VX, h, Wh, Z = self.forward(X=X, train=False, returnIntermediate=True)
    dZ = self.callActivation(Wh, self.outact, prime=True)
    dLdZ = dL * dZ

    # calculate gradJ_w
    dJ_w = dLdZ * np.tile(h.T, (self.W.shape[0],1))

    # backpropogation step
    sum_dZ = np.dot(dLdZ.T, self.W).T
    dH = self.callActivation(VX, self.hiddenact, prime=True)
    dHsum_dZ = sum_dZ[:self.V.shape[0],:] * dH

    # calculate gradJ_v
    dJ_v = dHsum_dZ * np.tile(X.T, (self.V.shape[0], 1))
    return dJ_w, dJ_v

def train(self, images, labels, learnRate=0.1, costFunc='meansq',
          bias=True, saveCost=1000, saveEvery=20000, hidact='tanh',
          outact='sig', wtvar=0.1, beta=0.1, continueTrain=False):
    if not continueTrain:
        self.X, self.Y, self.bias, self.wtvar = images, labels, bias, wtvar
        self.hiddenact, self.outact = hidact, outact
        self.learnRate, self.startLR  = learnRate, learnRate
        self.costs, self.errrate = [], [] # to keep track of costs/error rate per iteration
        self.numIter = 0 # initialize number of iterations for training purposes

        addUnit = 0
        if bias:
            self.X = np.vstack((self.X, np.ones((1,self.X.shape[1]))))
            addUnit = 1

        self.V = np.random.randn(self.numHiddenUnits, self.numInput + addUnit)
        * np.sqrt(wtvar) # input to hidden layer
        self.W = np.random.randn(self.numOutput, self.numHiddenUnits + addUnit)
        * np.sqrt(wtvar) # hidden to output layer
```

```
            self.inds = np.random.permutation(self.X.shape[1]) # shuffle the samples
            # current sample we're at, current epoch we're at
            self.currind, self.currepoch = 0, 1

        try:
            while True:
                # save weights saveEvery iterations
                if self.numIter % saveEvery == 0:
                    self.saveWeights(costFunc=costFunc)

                # calculate the gradients
                samp = self.X[:,self.inds[self.currind]]
                label = self.Y[:,self.inds[self.currind]]
                gradW, gradV = self.backwardProp(X=np.tile(samp, (1,1)).T,
                Y=np.tile(label, (1,1)).T, cost=costFunc)

                if self.numIter == 0:
                    deltV = - self.learnRate * gradV
                    deltW = - self.learnRate * gradW

                # update
                self.W = self.W + deltW
                self.V = self.V + deltV
                deltW = - self.learnRate * gradW + beta * deltW
                deltV = - self.learnRate * gradV + beta * deltV

                # print out cost on each saveCost number of iterations
                if self.numIter % saveCost == 0:
                    cost = self.callCost(costFunc=costFunc)
                    # estCost = self.numericalGradient(np.tile(samp, (1,1)).T,
                    np.tile(label, (1,1)).T, costFunc=costFunc)
                    self.costs.append(cost)
                    self.errrate.append(self.benchmarck(self.predict(images)))
                    print 'Cost is {0}...'.format(cost)
                    # print 'Numerical gradient estimate for dJ/dV_3 is {0}
                    compare to {1}...'.format(estCost, np.mean(gradV[10,10]))

                self.numIter += 1 # keep track of iterations

                # update currind correctly
                if self.currind == self.X.shape[1] - 1:
                    print 'Epoch {0} finished'.format(self.currepoch)
                    self.currind, self.currepoch = 0, self.currepoch + 1
                    self.assignLearningRate() # change the learning rate
                else:
                    self.currind += 1
        except KeyboardInterrupt:
            return

    def predict(self, images):
        testIms = images
        if self.bias:
            testIms = np.vstack((testIms, np.ones((1, testIms.shape[1]))))
        labels =  self.forward(X=testIms, train=False)
        return np.argmax(labels, axis=0) # pick the label that is the highest value
```

The following is the preprocessing methods.

```
def normalize(X, scalefactor=1):
```

```
    Xmean = np.mean(X, axis=1)
    Xhat = X - matlib.repmat(np.tile(Xmean, (1,1)).T, 1, X.shape[1])
    Xstd = np.std(Xhat, axis=1)
    Xhat = (Xhat / matlib.repmat(np.tile(Xstd, (1,1)).T, 1, X.shape[1])) * scalefactor
    return np.nan_to_num(Xhat), Xmean, Xstd

def binarize(X):
    return (X > 0) * 1
```

The following is the script used to train the mean-squared error neural network. All other networks were trained using similar scripts but with the parameters adjusted according to the values presented in the sections above.

```
inputsize = Xtrain_norm.shape[0]
outputsize = Ytrain.shape[0]
costFunc = 'meansq'
learnRate = 0.1 # gradient descent learning rate
wtvar = 0.01**2 # for initializing the weights, STD of 0.01
beta = 0.01 # momentum velocity factor
saveCost = 1000 # how often to save cost
saveEvery = 30000 # how often to save weights

nn = NeuralNetwork.NeuralNetwork(inputsize, outputsize)

start_time = time.time()

nn.train(Xtrain_norm, Ytrain, costFunc=costFunc, learnRate=learnRate, wtvar=wtvar, beta=beta,
saveCost=saveCost, saveEvery=saveEvery)

end_time = time.time()

print 'Training time was {0} seconds....'.format(end_time - start_time)

plt.figure()
plt.plot(1000*np.arange(len(nn.costs)), nn.costs)
plt.title('Cost per 1000 iterations')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.show()

plt.figure()
plt.plot(1000*np.arange(len(nn.errrate)), [elem/Xtrain_norm.shape[1] for elem in nn.errrate])
plt.title('Training error per 1000 iterations')
plt.xlabel('Iteration')
plt.ylabel('Error')
plt.show()

# final training error
print nn.errrate[-1]/50000

Yvalidpred = nn.predict(Xvalid_norm)

# to calculate validation error
wrong = np.sum(np.argmax(Yvalid, axis=0) != Yvalidpred)
```

References

[1] Momentum and Learning Rate Adaptation
https://www.willamette.edu/ gorr/classes/cs449/momrate.html