UC BERKELEY CS

189 Neural Networks HW4

Max Johansen

November 3, 2016

1 BACKPROP DERIVATION

Let $l^{[0]}$ represent the input layer, $l^{[1]}$ represent the hidden layer and $l^{[2]}$ the output layer. Let the activation (activation function applied to weighted combination of inputs) at layer $l^{[k]}$ be represented by $\alpha^{[k]}$. Let $W_{ij}^{[k]}$ represent the weight from node $j \in l^{[k-1]}$ to $i \in l^{[k]}$. Let $g^{[k]}$ represent the activation function for $l^{[k]}$. Let $z_i^{[k]}$ represent the weighted combination of inputs to node $i \in l^{[k]}$.

$$z_i^{[k]} = \sum_{j=1} W_{ij}^{[k]} \alpha^{[k-1]} \tag{1.1}$$

$$\begin{aligned} \alpha^{[k]} &= g^{[k]}(W^{[k]}\alpha^{[k-1]}) \\ &= g^{[k]}(z^{[k]}) \end{aligned} \tag{1.2}$$

1.1 COST FUNCTION

We use cross entropy to represent cost J. Let $n_i(x)$ represent the ith component of the network output given features x.

$$J = -\sum_{i=1}^{n_{out}} y_i ln(n_i(x)) \tag{1.3}$$

We need to compute $\frac{\partial J}{\partial W_{ij}}$ using the chain rule.

$$\frac{\partial J}{\partial W_{ij}} = \frac{\partial J}{\partial z_i^{[2]}} \times \frac{\partial z_i^{[2]}}{\partial W_{ij}} \tag{1.4}$$

$$\frac{\partial z_i^{[2]}}{\partial W_{ij}} = \alpha_j^{[1]} \tag{1.5}$$

$$\begin{aligned}
\frac{\partial J}{\partial z_i^{[2]}} &= \frac{\partial - \sum_{j=1}^{n_{out}} y_j ln(n_j(x))}{\partial z_i^{[2]}} \\
&= \frac{\partial - \sum_{j=1}^{n_{out}} y_j ln(g^{[2]}(z_j^{[2]}))}{\partial z_i^{[2]}} \\
&= -\sum_{j=1}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} \frac{\partial g^{[2]}(z_j^{[2]})}{\partial z_i^{[2]}} \\
&= -\sum_{j=i}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} \frac{\partial g^{[2]}(z_j^{[2]})}{\partial z_i^{[2]}} - \sum_{j\neq i}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} \frac{\partial g^{[2]}(z_j^{[2]})}{\partial z_i^{[2]}} \\
&= -\frac{y_i}{g^{[2]}(z_i^{[2]})} g^{[2]}(z_i^{[2]})(1 - g^{[2]}(z_i^{[2]})) - \sum_{j\neq i}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} \frac{\partial g^{[2]}(z_j^{[2]})}{\partial z_i^{[2]}} \\
&= -y_i(1 - g^{[2]}(z_i^{[2]})) - \sum_{j\neq i}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} \frac{\partial g^{[2]}(z_j^{[2]})}{\partial z_i^{[2]}} \\
&= -y_i(1 - g^{[2]}(z_i^{[2]})) + \sum_{j\neq i}^{n_{out}} \frac{y_j}{g^{[2]}(z_j^{[2]})} g^{[2]}(z_j^{[2]}) g^{[2]}(z_i^{[2]}) \\
&= -y_i(1 - g^{[2]}(z_i^{[2]})) + \sum_{j\neq i}^{n_{out}} y_j g^{[2]}(z_i^{[2]}) \\
&= -y_i + g^{[2]}(z_i^{[2]}) \sum_{j=1}^{n_{out}} y_j \\
&= -y_i + g^{[2]}(z_i^{[2]}) \\
&= g^{[2]}(z_i^{[2]}) - y_i
\end{aligned} \tag{1.6}$$

Here's some more notation

$$\delta_i^{[k]} = \frac{\partial J}{\partial z_i^{[k]}} \tag{1.7}$$

This implies

$$\delta_i^{[2]} = g^{[2]}(z_i^{[2]}) - y_i \tag{1.8}$$

Vectorized

$$\delta^{[2]} = g^{[2]}(z^{[2]}) - y \tag{1.9}$$

We can use induction and chain rule here to help us out.

$$\begin{aligned}
\delta_i^{[k-1]} &= \frac{\partial J}{\partial z_i^{[k-1]}} \\
&= \sum_j \frac{\partial J}{\partial z_j^{[k]}} \frac{\partial z_j^{[k]}}{\partial \alpha_i^{[k-1]}} \frac{\partial \alpha_i^{[k-1]}}{\partial z_i^{[k-1]}} \\
&= g^{[k-1]'} z_i^{[k-1]} \sum_j \delta_j^{[k]} W_{ji}^{[k]}
\end{aligned} \tag{1.10}$$

Vectorized

$$\delta^{[k-1]} = g^{[k-1]'} z^{[k-1]} \odot W^{[k]T} \delta^{[k]} \tag{1.11}$$

Now, we combine $\frac{\partial J}{\partial z_i^{[k]}} \times \frac{\partial z_i^{[k]}}{\partial W_{ij}}$ to find $\frac{\partial J}{\partial W_{ij}^{[k]}}$

$$\frac{\partial J}{\partial W_{ij}^{[k]}} = \delta_i^{[k]} \alpha_j^{[k-1]} \tag{1.12}$$

Matrix notation

$$\frac{\partial J}{\partial W^{[k]}} = \delta^{[k]} \alpha^{[k-1]T} \tag{1.13}$$

Now we derive $\frac{\partial J}{\partial V}$ $(W^{[1]} = V)$ (x represents input to network).

$$\begin{aligned}
\frac{\partial J}{\partial W^{[1]}} &= \delta^{[1]} \alpha^{[0]T} \\
&= \delta^{[1]} x^T \\
&= g^{[1]'}(z^{[1]}) \odot W^{[2]T} \delta^{[2]} x^T \\
&= g^{[1]'}(z^{[1]}) \odot W^{[2]T} (g^{[2]}(z^{[2]}) - y) x^T
\end{aligned} \tag{1.14}$$

Now we explicitly state $\frac{\partial J}{\partial W^{[2]}}$.

$$\begin{aligned}
\frac{\partial J}{\partial W^{[2]}} &= \delta^{[2]} \alpha^{[1]T} \\
&= (g^{[2]}(z^{[2]}) - y) \alpha^{[1]T}
\end{aligned} \tag{1.15}$$

Now we define the SGD update rule for $W^{[k]}$. Let γ represent the learning rate.

$$W_{t+1}^{[k]} = W_t^{[k]} - \gamma \frac{\partial J}{\partial W^{[k]}} \tag{1.16}$$

Update rule for $W = W^{[2]}$

$$\begin{aligned} W^{[2]}_{t+1} &= W^{[2]}_t - \gamma \frac{\partial J}{\partial W^{[2]}} \\ &= W^{[2]}_t - \gamma (g^{[2]}(z^{[2]}) - y) \alpha^{[1]T} \end{aligned} \tag{1.17}$$

Update rule for $V = W^{[1]}$

$$\begin{aligned} W^{[1]}_{t+1} &= W^{[1]}_t - \gamma \frac{\partial J}{\partial W^{[1]}} \\ &= W^{[1]}_t - \gamma g^{[1]'}(z^{[1]}) \odot W^{[2]T}(g^{[2]}(z^{[2]}) - y) \end{aligned} \tag{1.18}$$

1.2 Implementation notes

I start with an initial learning rate of $1e{-}3$ and decay by $1/2$ every epoch. I stopped training after 5 epochs. Total train time was roughly 20 minutes. I initialized the weights to be zero mean gaussians normalized by $sqrt(n_{entries}$. I arrived at trainingAccuracy 0.9966 validationAccuracy 0.9397 which could absolutely be improved with regularization and the addition of a conv layer.

1.3 Kaggle score

My Kaggle score was 0.94580, position 210.

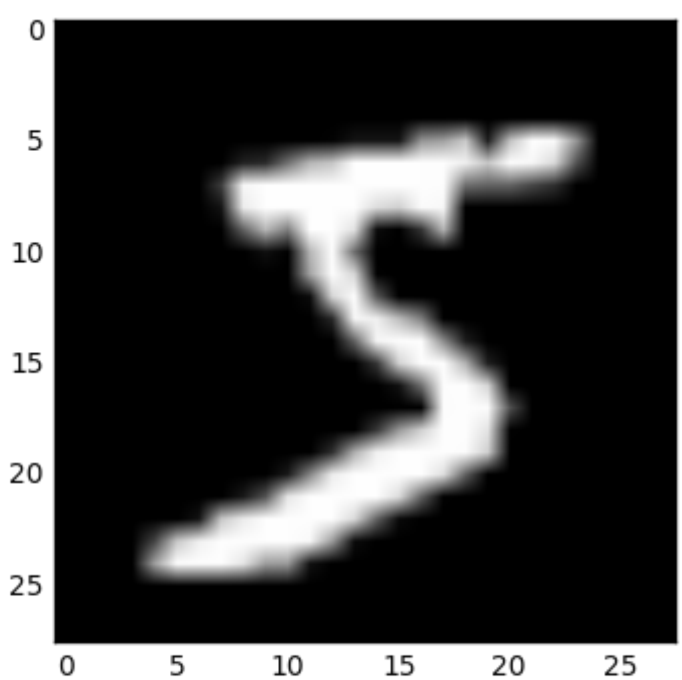
In [1]:
```
from mnist import MNIST
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from scipy import exp, clip
# Allowed on piazza
from sklearn.utils import shuffle
from numpy.random import randint
%matplotlib inline
```

In [22]:
```
def load_dataset():
    mndata = MNIST('./data/')
    X_train, labels_train = map(np.array, mndata.load_training())
    # The test labels are meaningless,
    # since you're replacing the official MNIST test set with our own te
st set
    X_test, _ = map(np.array, mndata.load_testing())
    # Remember to center and normalize the data...
    return X_train, labels_train, X_test

X_train, labels_train, X_test = load_dataset()
X_train = (X_train - np.mean(X_train, axis=0)) / 255
X_test = (X_test - np.mean(X_test, axis=0)) / 255
```

In [3]:
```
plt.imshow(X_train[0].reshape((28,28)), cmap='Greys_r')
```

Out[3]: <matplotlib.image.AxesImage at 0x1126de860>



In [4]:
```
def safe_exp(x):
    """to avoid inf and NaN values"""
    """https://github.com/miha-stopar/nnets/blob/master/neuron/tools.p
y"""
    return exp(clip(x, -500, 500))
def layerOneActivations(z):
    return np.maximum(z, 0)
def layerOneDeriv(z):
    return np.where(z > 0, 1,0)
def layerTwoActivations(x):
    e_x = safe_exp(x - np.max(x))
    return e_x / (e_x).sum(axis=0)
def layerTwoDeriv(layerTwoActivation, y):
    return layerTwoActivation - y
def one_hot(labels_train):
    '''Convert categorical labels 0,1,2,....9 to standard basis vectors
 in R^{10} '''
    return np.eye(10)[labels_train]
def computeCostAndAccuracy(inputDataset, labels):
    inputDataset = np.hstack((inputDataset, np.ones((inputDataset.shape[0
1))))
    hiddenLayerWeightedInput = weights[0].dot(inputDataset.T)
    hiddenLayerActivation = np.vstack((g[0](hiddenLayerWeightedInput), n
p.ones((1,hiddenLayerWeightedInput.shape[1]))))

    outputLayerWeightedInput = weights[1].dot(hiddenLayerActivation)
    outputLayerActivation = g[1](outputLayerWeightedInput).T

    #digitPredictions = np.diag(np.array([range(10)])).dot(outputLayerAc
tivation)
    digitPredictions = np.argmax(outputLayerActivation, axis=1)
    labelsConverted = np.argmax(labels, axis=1)
    theCost = -(outputLayerActivation * np.log(labels + 1e-200)).sum()
    accuracy = np.sum(labelsConverted==digitPredictions) / len(labelsCon
verted)
    return theCost, accuracy
```

In [42]:

```
n_in = 784
n_hid = 200
n_out = 10
n_layers = 3

# Param for layer k exists in array position k-1
# i.e. [V,W]
g = [layerOneActivations, layerTwoActivations]
g_prime = [layerOneDeriv, layerTwoDeriv]
weights = [np.random.randn(n_hid, n_in+1) * np.sqrt(2/n_hid),
np.random.randn(n_out, n_hid+1) * np.sqrt(2/n_out)]

train_size = 50000
validate_size = 10000
labels, inputData = shuffle(one_hot(labels_train), X_train)
# 1 epoch approx 30 sec
# num_iters = int(40 * train_size)
num_iters = train_size * 10
# compute_cost_interval = int(np.sqrt(num_iters))
compute_cost_interval = 10000

trainLabels, trainData = labels[train_size:], inputData[train_size:]
validateLabels, validateData = labels[:validate_size], inputData[:valida
te_size]

# Save costs here
trainingCostList, validationCostList = np.array([]), np.array([])
trainingAccuracyList, validationAccuracyList = np.array([]),
np.array([])

# Hyperparameters
initialLearningRate = 1e-3
decayFactor = (initialLearningRate / num_iters) * 1e-2
momentumFactor = 0
prevDw = 0
prevDv = 0
learningRate = 2*initialLearningRate
for i in range(num_iters):
    if i % train_size == 0:
        learningRate /= 2

    # Select a random point
    datapointIndex = randint(0, len(trainLabels))
    inputVector = np.append(trainData[datapointIndex], [1])
    labelVector = trainLabels[datapointIndex]

    # Feed forward
    hiddenLayerWeightedInput = weights[0].dot(inputVector)
    hiddenLayerActivation = np.append(g[0](hiddenLayerWeightedInput),
[1])

    outputLayerWeightedInput = weights[1].dot(hiddenLayerActivation)
    outputLayerActivation = g[1](outputLayerWeightedInput)

    # Compute cost (don't do every time)
    if i % compute_cost_interval == 0:
        tCost, tAccuracy = computeCostAndAccuracy(trainData,
```

```
trainLabels)
        trainingCostList = np.append(trainingCostList, tCost)
        trainingAccuracyList = np.append(trainingAccuracyList,
tAccuracy)

        vCost, vAccuracy = computeCostAndAccuracy(validateData, validate
Labels)
        validationCostList = np.append(validationCostList, vCost)
        validationAccuracyList = np.append(validationAccuracyList, vAccu
racy)

    # Update outer weights
    # W
    deltaW = g_prime[1](outputLayerActivation, labelVector)
    dW = np.outer(deltaW,hiddenLayerActivation) + momentumFactor * prevD
w
    prevDw = dW

    # V
    deltaV = g_prime[0](hiddenLayerWeightedInput) * weights[1].T.dot(del
taW)[:-1]
    dV = np.outer(deltaV, inputVector) + momentumFactor * prevDv
    prevDv = dV

    # Update weights
    # W
    weights[1] -= learningRate*dW
    # V
    weights[0] -= learningRate*dV
```

In [43]:
```
# Plot the cost over the iterations
plt.figure(1)
# plt.subplot(211)
validationCostPlot = plt.plot(validationCostList, label="validation cos
t")[0]
trainingCostPlot = plt.plot(trainingCostList, label="training cost")[0]
plt.xlabel('iteration x' + str(compute_cost_interval))
plt.ylabel('$J$', fontsize=15)
plt.title('Cost over iteration')
plt.grid()
plt.legend()
# Plot the accuracy over the iterations
# plt.subplot(212)
plt.figure(2)
validationAccuracyPlot = plt.plot(validationAccuracyList, label="validat
ion accuracy")[0]
trainingAccuracyPlot = plt.plot(trainingAccuracyList, label="training ac
curacy")[0]
plt.xlabel('iteration x' + str(compute_cost_interval))
plt.ylabel('$J$', fontsize=15)
plt.title('Accuracy over iteration')
plt.grid()
plt.legend()
print("trainingAccuracy",trainingAccuracyList[-1], "validationAccuracy",
 validationAccuracyList[-1])
```

```
trainingAccuracy 0.9966 validationAccuracy 0.9397
```