CS 224N:Assignment #1

WeiYang

1. Softmax

a)

$$\text{softmax}(x+c)_i = \frac{\exp(x_i+c)}{\sum_{j=1}^{\dim(x)} \exp(x_j+c)} = \frac{\exp(c)\exp(x_i)}{\exp(c)\sum_{j=1}^{\dim(x)} \exp(x_j)} = \frac{\exp(x_i)}{\sum_{j=1}^{\dim(x)} \exp(x_j)} = \text{softmax}(x)_i$$

b)

```
def softmax(x):
    orig_shape = x.shape
    if len(x.shape) > 1:
        ### YOUR CODE HERE
        x -= np.max(x, axis=1, keepdims=True)
        x = np.exp(x) / np.sum(np.exp(x), axis=1,
keepdims=True)
        ### END YOUR CODE
    else:
        ### YOUR CODE HERE
        x -= np.max(x)
        x = np.exp(x) / np.sum(np.exp(x))
        ### END YOUR CODE
    assert x.shape == orig_shape
    return x
```

2. Neural Network Basics

a)

$$\sigma'(x) = \left(\frac{1}{1+e^{-x}}\right)'$$
$$= -(\frac{1}{1+e^{-x}})^2 \cdot (e^{-x})'$$
$$= \frac{e^{-x}}{(1+e^{-x})^2}$$
$$= \frac{1}{1+e^{-x}} \cdot (1 - \frac{1}{1+e^{-x}})$$
$$= \sigma(x) \cdot (1 - \sigma(x))$$

b)

$$\frac{\partial CE(y,\hat{y})}{\partial \theta_i} = \frac{\partial - \sum_i y_i \log(\hat{y}_i)}{\partial \theta_i}$$
$$= \frac{\partial - \log(\hat{y}_k)}{\partial \theta_i}$$
$$= -\frac{\partial \log(s(\theta_k))}{\partial \theta_i}$$
$$= \begin{cases} \hat{y}_i - 1 & i = k \\ \hat{y}_i & i \neq k \end{cases}$$

c)

$$let\ z_2 = hW_2 + b_2, z_1 = xW_1 + b_1$$
$$\delta_1 = \frac{\partial CE}{\partial z_2} = \hat{y} - y$$
$$\delta_2 = \frac{\partial CE}{\partial h} = \delta_1 \frac{\partial z_2}{\partial h} = \delta_1 W_2^T$$
$$\delta_3 = \frac{\partial CE}{\partial z_1} = \delta_2 \frac{\partial h}{\partial z_1} = \delta_2 \circ \sigma'(z_1)$$
$$\frac{\partial CE}{\partial x} = \delta_3 \frac{\partial z_1}{\partial x} = \delta_3 W_1^T$$

d)

$$(D_x + 1)H + (H + 1)D_y$$

e)

```
def sigmoid(x):
    s = 1.0 / (1.0 + np.exp(-x))
    return s

def sigmoid_grad(s):
    ds = s * (1.0 - s)
    return ds
```

f)

```
def gradcheck_naive(f, x):
   ### ...
   ### YOUR CODE HERE:
   old_val = x[ix]
   x[ix] = old_val - h
   random.setstate(rndstate)
   fxh1, _ = f(x)
   x[ix] = old_val + h
   random.setstate(rndstate)
   fxh2, _ = f(x)
   numgrad = (fxh2 - fxh1) / (2.0 * h)
   x[ix] = old_val
   ### END YOUR CODE
   ### ...
```

g)

```
def forward_backward_prop(data, labels, params,
dimensions):
    ### Unpack network parameters (do not modify)
    ofs = 0
    Dx, H, Dy = (dimensions[0], dimensions[1],
dimensions[2])

    W1 = np.reshape(params[ofs:ofs+ Dx * H], (Dx, H))
    ofs += Dx * H
    b1 = np.reshape(params[ofs:ofs + H], (1, H))
    ofs += H
    W2 = np.reshape(params[ofs:ofs + H * Dy], (H, Dy))
    ofs += H * Dy
```

```
    b2 = np.reshape(params[ofs:ofs + Dy], (1, Dy))

    ### YOUR CODE HERE: forward propagation
    N = data.shape[0]
    h = sigmoid(data.dot(W1) + b1)
    output = softmax(h.dot(W2) + b2)
    cost = - np.sum(np.log(output[labels == 1])) / N
    ### END YOUR CODE

    ### YOUR CODE HERE: backward propagation
    grad_output = output - labels
    gradW2 = np.dot(h.T, grad_output) / N
    gradb2 = np.sum(grad_output, axis=0, keepdims=True) / N
    grad_h = np.dot(grad_output, W2.T) * sigmoid_grad(h)
    gradW1 = np.dot(data.T, grad_h) / N
    gradb1 = np.sum(grad_h, axis=0, keepdims=True) / N
    ### END YOUR CODE

    ### Stack gradients (do not modify)
    grad = np.concatenate((gradW1.flatten(),
gradb1.flatten(),
        gradW2.flatten(), gradb2.flatten()))
    return cost, grad
```

3. word2vec

a)

$$\frac{\partial J}{\partial v_c} = \frac{\partial J}{\partial U^T v_c} \cdot \frac{\partial U^T v_c}{\partial v_c} = (\hat{y} - y)U$$

b)

$$\frac{\partial J}{\partial U} = \frac{\partial J}{\partial U^T v_c} \cdot \frac{\partial U^T v_c}{\partial U} = (\hat{y} - y)v_c$$

c)

$$\frac{\partial J}{\partial v_c} = (\sigma(u_o{}^T v_c) - 1)u_o - \sum_{k=1}^{K}(\sigma(-u_k{}^T v_c) - 1)u_k$$

$$\frac{\partial J}{\partial u_o} = (\sigma(u_o{}^T v_c) - 1)v_c$$

$$\frac{\partial J}{\partial u_k} = -(\sigma(-u_k{}^T v_c) - 1)v_c, \text{ for all } k = 1, 2, \ldots, K$$

d)

$$\frac{\partial J_{skip-gram}(word_{c-m\ldots c+m})}{\partial U} = \sum_{-m \le j \le m, j \ne 0} \frac{\partial F(w_{c+j}, v_c)}{\partial U}$$

$$\frac{\partial J_{skip-gram}(word_{c-m\ldots c+m})}{\partial v_c} = \sum_{-m \le j \le m, j \ne 0} \frac{\partial F(w_{c+j}, v_c)}{\partial v_c}$$

$$\frac{\partial J_{skip-gram}(word_{c-m\ldots c+m})}{\partial v_j} = 0$$

e)

```
def normalizeRows(x):
    ### YOUR CODE HERE
    k = np.sum(x * x, axis=1, keepdims=True) ** 0.5
    x /= k
    ### END YOUR CODE
    return x

def softmaxCostAndGradient(predicted, target,
outputVectors, dataset):
    probabilities = softmax(predicted.dot(outputVectors.T))
    cost = -np.log(probabilities[target])
    delta = probabilities
    delta[target] -= 1
    N = delta.shape[0]
    D = predicted.shape[0]
    grad = delta.reshape((N,1)) * predicted.reshape((1,D))
    gradPred =
(delta.reshape((1,N)).dot(outputVectors)).flatten()
    return cost, gradPred, grad

def negSamplingCostAndGradient(predicted, target,
```

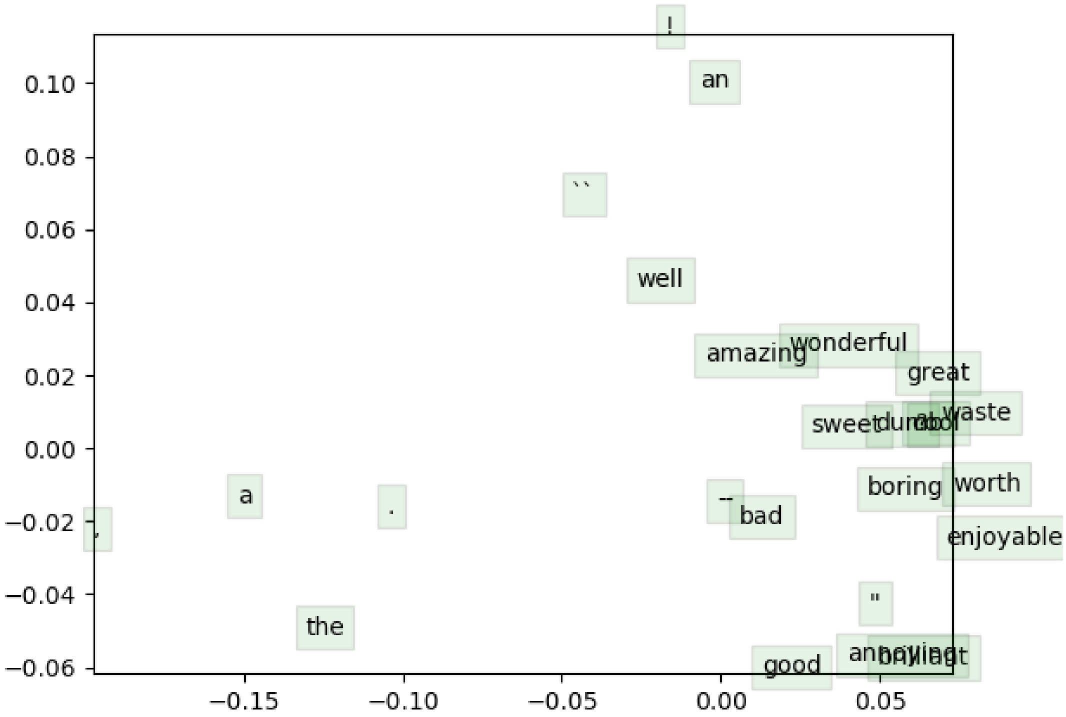
```
outputVectors, dataset,
                            K=10):
    indices = [target]
    indices.extend(getNegativeSamples(target, dataset, K))
    ### YOUR CODE HERE
    grad = np.zeros(outputVectors.shape)
    gradPred = np.zeros(predicted.shape)
    cost = 0
    z = sigmoid(np.dot(outputVectors[target], predicted))
    cost -= np.log(z)
    grad[target] += predicted * (z - 1.0)
    gradPred += outputVectors[target] * (z - 1.0)
    for k in indices[1:]:
        z = sigmoid(np.dot(outputVectors[k], predicted))
        cost -= np.log(1.0 - z)
        grad[k] += predicted * z
        gradPred += outputVectors[k] * z
    ### END YOUR CODE
    return cost, gradPred, grad

def skipgram(currentWord, C, contextWords, tokens,
inputVectors, outputVectors,
             dataset,
word2vecCostAndGradient=softmaxCostAndGradient):
    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)
    ### YOUR CODE HERE
    currentI = tokens[currentWord]
    predicted = inputVectors[currentI, :]
    for cwd in contextWords:
        idx = tokens[cwd]
        cc, gp, gg = word2vecCostAndGradient(predicted, idx,
outputVectors, dataset)
        cost += cc
        gradOut += gg
        gradIn[currentI] += gp
    ### END YOUR CODE
    return cost, gradIn, gradOut
```

f)

```
def sgd(f, x0, step, iterations, postprocessing=None,
useSaved=False,
        PRINT_EVERY=10):
```

```
### ...
### YOUR CODE HERE
cost, grad = f(x)
x = x - step * grad
x = postprocessing(x)
### END YOUR CODE
### ...
```

g)



h)

```
def cbow(currentWord, C, contextWords, tokens,
inputVectors, outputVectors,
        dataset,
word2vecCostAndGradient=softmaxCostAndGradient):
    cost = 0.0
    gradIn = np.zeros(inputVectors.shape)
    gradOut = np.zeros(outputVectors.shape)
    ### YOUR CODE HERE
    D = inputVectors.shape[1]
    predicted = np.zeros((D,))
    indices = [tokens[cwd] for cwd in contextWords]
    for idx in indices:
```

```
        predicted += inputVectors[idx, :]
    cost, gp, gradOut = word2vecCostAndGradient(predicted,
tokens[currentWord], outputVectors, dataset)
    gradIn = np.zeros(inputVectors.shape)
    for idx in indices:
        gradIn[idx] += gp
    ### END YOUR CODE
    return cost, gradIn, gradOut
```

4. Sentiment Analysis

a)

```
def getSentenceFeatures(tokens, wordVectors, sentence):
    sentVector = np.zeros((wordVectors.shape[1],))
    ### YOUR CODE HERE
    indices = [tokens[word] for word in sentence]
    sentVector = np.mean(wordVectors[indices], axis=0)
    ### END YOUR CODE
    assert sentVector.shape == (wordVectors.shape[1],)
    return sentVector
```

b) avoid overfitting

c)

```
def getRegularizationValues():
    values = None   # Assign a list of floats in the block
below
    ### YOUR CODE HERE
    values = np.logspace(-4, 2, num=20, base=10)
    ### END YOUR CODE
    return sorted(values)
```

d)

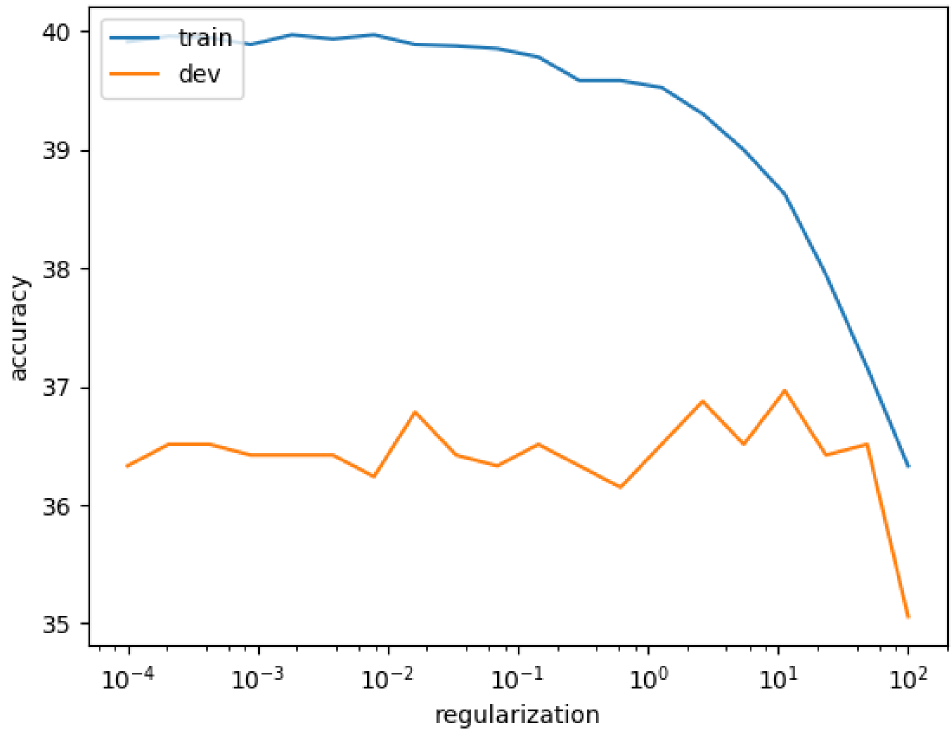
pretrained:

```
=== Recap ===
Reg             Train   Dev     Test
1.00E-04        39.911  36.331  37.014
2.07E-04        39.958  36.512  36.968
4.28E-04        39.946  36.512  36.968
8.86E-04        39.888  36.421  37.059
1.83E-03        39.970  36.421  36.968
3.79E-03        39.934  36.421  37.149
7.85E-03        39.970  36.240  37.149
1.62E-02        39.888  36.785  37.285
3.36E-02        39.876  36.421  37.466
6.95E-02        39.853  36.331  37.195
1.44E-01        39.782  36.512  37.511
2.98E-01        39.583  36.331  37.285
6.16E-01        39.583  36.149  37.285
1.27E+00        39.525  36.512  37.330
2.64E+00        39.302  36.876  37.240
5.46E+00        38.998  36.512  37.376
1.13E+01        38.624  36.966  37.466
2.34E+01        37.945  36.421  36.923
4.83E+01        37.161  36.512  36.154
1.00E+02        36.330  35.059  35.701

Best regularization value: 1.13E+01
Test accuracy (%): 37.466063
```

yourvectors:

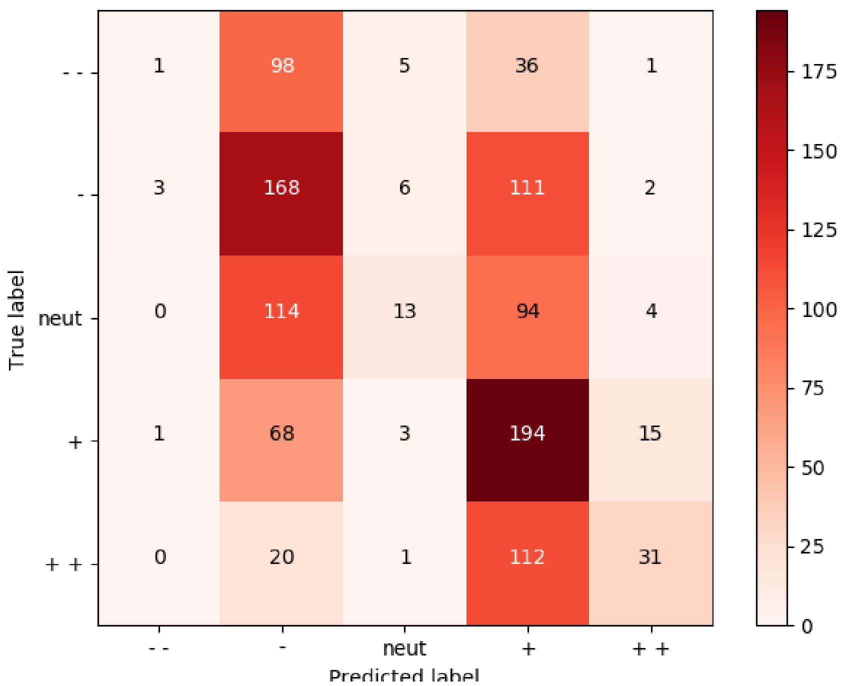
```
=== Recap ===
Reg             Train   Dev     Test
1.00E-04        30.969  32.698  30.362
2.07E-04        30.993  32.607  30.317
4.28E-04        31.063  32.425  30.407
8.86E-04        31.180  32.698  30.317
1.83E-03        31.133  32.153  30.362
3.79E-03        31.039  32.243  30.407
7.85E-03        30.946  32.334  30.226
1.62E-02        30.770  32.243  29.955
3.36E-02        30.431  31.608  29.955
6.95E-02        30.384  32.243  30.045
1.44E-01        30.150  31.880  29.683
2.98E-01        29.459  31.880  28.959
6.16E-01        29.354  31.063  27.964
1.27E+00        28.652  28.429  26.244
2.64E+00        27.926  26.703  24.525
5.46E+00        27.317  25.704  23.213
1.13E+01        27.235  25.522  23.077
2.34E+01        27.235  25.522  23.032
4.83E+01        27.247  25.522  23.032
1.00E+02        27.247  25.522  23.032

Best regularization value: 1.00E-04
Test accuracy (%): 30.361991
```

e)



train
dev
accuracy
40
39
38
37
36
35
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
regularization

f)



True label
Predicted label
- -
-
neut
+
+ +
1 98 5 36 1
3 168 6 111 2
0 114 13 94 4
1 68 3 194 15
0 20 1 112 31
175
150
125
100
75
50
25
0