TCSS 455 Machine Learning

Homework #3

Tianyi Li
Student ID: 1827924

1. <u>Backpropagation</u>:

Consider a two-layer feedforward ANN with two inputs a and b, one hidden unit c, and one output unit d. This network has five weights (w_{ca}, w_{cb}, w_{c0}, w_{dc}, w_{d0}), where w,o represents the threshold weight for unit x. Initialize these weights to the values (0.1, 0.1, 0.1, 0.1, 0.1), then give their values after each of the first two training iterations of the Backpropagation algorithm. Assume learning rate $\eta = 0.3$, momentum $\alpha = 0.3$, incremental weight updates, and the following training examples:

a	b	d
1	0	1
0	1	0

The network:

The equation for the sigmoid activation function: $\sigma(y) = \frac{1}{1+e^{-y}}$

<u>Training example #1</u>: *With* $a = 1, b = 0$, *calculate the outputs,*

$$O_c = \sigma(w_{ca} \times a + w_{cb} \times b + w_{c0}) = \sigma(0.1 \times 1 + 0.1 \times 0 + 0.1)$$

$$= \sigma(0.2) = \frac{1}{1 + e^{-0.2}} = 0.5498$$

$$O_d = \sigma(w_{dc} \times c + w_{d0}) = \sigma(0.1 \times 0.5498 + 0.1)$$

$$= \sigma(0.15498) = \frac{1}{1 + e^{-0.15498}} = 0.5387$$

With $d = 1$, *calculate the error of the hidden layer,*

$$\delta_d = O_d \times (1 - O_d) \times (d - O_d)$$

$$= 0.5387 \times (1 - 0.5387) \times (1 - 0.5387) = 0.1146$$

$$\delta_c = O_c \times (1 - O_c) \times (w_{dc} - O_d)$$

$$= 0.5498 \times (1 - 0.5498) \times 0.1 \times 0.1146 = 0.002837$$

Hence, calculate the error terms, with $a = 1, b = 0$ *and* $\eta = 0.3$,

$$\Delta w_{ca}(1) = \eta \times O_c \times a = 0.3 \times 0.002837 \times 1 = 0.0008511$$

$$\Delta w_{cb}(1) = \eta \times O_c \times b = 0.3 \times 0.002837 \times 0 = 0$$

$$\Delta w_{c0}(1) = \eta \times O_c \times O = 0.3 \times 0.002837 \times 1 = 0.0008511$$

$$\Delta w_{dc}(1) = \eta \times O_d \times c = 0.3 \times 0.1146 \times 0.5498 = 0.01890$$

$$\Delta w_{do}(1) = \eta \times O_d \times O = 0.3 \times 0.1146 \times 1 = 0.03438$$

Therefore, the new weights are:

$$w_{ca} = w_{ca} + \Delta w_{ca}(1) = 0.1 + 0.0008511 = 0.1009$$

$$w_{cb} = w_{cb} + \Delta w_{cb}(1) = 0.1 + 0 = 0.1$$

$$w_{c0} = w_{c0} + \Delta w_{co}(1) = 0.1 + 0.0008511 = 0.1009$$

$$w_{dc} = w_{do} + \Delta w_{dc}(1) = 0.1 + 0.01890 = 0.1189$$

$$w_{do} = w_{do} + \Delta w_{do}(1) = 0.1 + 0.03438 = 0.1344$$

<u>*Training example #2*</u>: *With $a = 0, b = 1$, calculate the outputs,*

$$O_c = \sigma(w_{ca} \times a + w_{cb} \times b + w_{c0}) = \sigma(0.1009 \times 0 + 0.1 \times 1 + 0.1009)$$

$$= \sigma(0.2009) = \frac{1}{1 + e^{-0.5501}} = 0.5501$$

$$O_d = \sigma(w_{dc} \times c + w_{d0}) = \sigma(0.1189 \times 0.5501 + 0.1344)$$

$$= \sigma(0.1998) = \frac{1}{1 + e^{-0.5498}} = 0.5498$$

With $d = 0$, calculate the error of the hidden layer,

$$\delta_d = O_d \times (1 - O_d) \times (d - O_d)$$

$$= 0.5498 \times (1 - 0.5498) \times (0 - 0.5498) = -0.1361$$

$$\delta_c = O_c \times (1 - O_c) \times (w_{dc} - O_d)$$

$$= 0.5501 \times (1 - 0.5501) \times 0.1189 \times -0.1361 = -0.004005$$

Hence, calculate the error terms, with $a = 1, b = 0, \eta = 0.3$, and $\alpha = 0.3$,

$$\Delta w_{ca}(2) = \eta \times O_c \times a + \alpha \times \Delta w_{ca}(1)$$

$$= 0.3 \times (-0.004005) \times 0 + 0.9 \times 0.0008511 = 0.0007660$$

$$\Delta w_{cb}(2) = \eta \times O_c \times b + \alpha \times \Delta w_{cb}(1)$$

$$= 0.3 \times (-0.004005) \times 1 + 0.9 \times 0 = -0.001202$$

$$\Delta w_{c0}(2) = \eta \times O_c \times O + \alpha \times \Delta w_{co}(1)$$

$$= 0.3 \times (-0.004005) \times 1 + 0.9 \times 0.0008511 = -0.0004355$$

$$\Delta w_{dc}(2) = \eta \times O_d \times c + \alpha \times \Delta w_{dc}(1)$$

$$= 0.3 \times (-0.1361) \times 0.5501 + 0.9 \times 0.01890 = -0.005451$$

$$\Delta w_{do}(2) = \eta \times O_d \times O + \alpha \times \Delta w_{do}(1)$$

$$= 0.3 \times (-0.1361) \times 1 + 0.9 \times 0.03438 = -0.009888$$

Therefore, the new weights are:

$$w_{ca} = w_{ca} + \Delta w_{ca}(2) = 0.1009 + 0.0007660 = 0.1017$$

$$w_{cb} = w_{cb} + \Delta w_{cb}(2) = 0.1 - 0.001202 = 0.09880$$

$$w_{c0} = w_{c0} + \Delta w_{co}(2) = 0.1009 - 0.0004355 = 0.1005$$

$$w_{dc} = w_{dc} + \Delta w_{dc}(2) = 0.1189 - 0.005451 = 0.1134$$

$$w_{do} = w_{do} + \Delta w_{do}(2) = 0.1344 - 0.009888 = 0.1245$$

2. Gradient Descent Weight Update Rule for a Tanh Unit:

Let us replace the sigmoid functionσin Figure 4.6 by the function "tanh". Derive the new weight update rule.

We are asked to assume the output of an unit to be $O = \tanh(\vec{w} \cdot \vec{x})$

$$= \tanh(w_0 x_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n).$$

The proof below are borrowing ideas from Chapter 4.5.3 from the textbook:

For each training example d*, every weight* ω_{ji} *is updated by adding to its* Δw_{ji}*:*

$$\Delta w_{ji} = -\eta \frac{\vartheta E_d}{\vartheta \omega_{ji}}$$

Where E_d *is the error on training example* d*, summed over all output unites in the network:*

$$E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - O_k)^2$$

- x_{ji} *= the i^th input to unit j*
- ω_{ji} *= the weight associated with the i^th input to unit j*
- $net_j = \sum \omega_{ji} x_{ji}$ *(the weighted sum of inputs for unit j)*
- o_j *= the output computed by unit j*
- t_j *= the target output for unit j*
- σ *= the sigmoid function*
- *outputs = the set of units in the final layer of the network*
- *Downstream(j) = the set of units whose immediate inputs include the output of unit j*

$$\frac{\vartheta E_d}{\vartheta \omega_{ji}} = \frac{\vartheta E_d}{\vartheta net_j} \cdot \frac{\vartheta net_j}{\vartheta \omega_{ji}} = \frac{\vartheta E_d}{\vartheta net_j} \cdot x_{ji}$$

Then, the training rule for output unit weights is

$$\frac{\vartheta E_d}{\vartheta net_j} = \frac{\vartheta E_d}{\vartheta o_j} \cdot \frac{\vartheta o_j}{\vartheta net_j}$$

Now, consider only the first term:

$$\frac{\vartheta E_d}{\vartheta o_j} = \frac{\vartheta}{\vartheta o_j} \cdot \frac{1}{2} \sum_{k \in outputs} (t_k - O_k)^2$$

The derivatives of $\frac{\vartheta}{\vartheta o_j}(t_k - O_k)^2$*will be 0 (zero)for an output unit* k*, except when* $k = j$*. Therefore we drop the summation over output unites and simply set* $k = j$*:*

$$\frac{\vartheta E_d}{\vartheta o_j} = \frac{\vartheta}{\vartheta o_j} \cdot \frac{1}{2} \left(t_j - o_j\right)^2$$

$$= \frac{1}{2} \times 2(t_j - o_j) \cdot \frac{\vartheta}{\vartheta o_j}(t_j - o_j)$$

$$= -(t_j - o_j)$$

Now, consider the second term:

$$\frac{\vartheta o_j}{\vartheta net_j} = \frac{\vartheta}{\vartheta net_j} \cdot \tanh(net_j)$$

$$= (1 - \tanh(net_j)^2) \cdot \frac{\vartheta}{\vartheta net_j}(net_j)$$

$$= (1 - o_j{}^2)$$

Hence, we combine both terms:

$$\frac{\vartheta E_d}{\vartheta net_j} = -(t_j - o_j) \cdot (1 - o_j{}^2)$$

Therefore,

$$\Delta w_{ji} = -\eta \cdot \frac{\vartheta E_d}{\vartheta \omega_{ji}}$$

$$= -\eta \cdot (-(t_j - o_j)) \cdot (1 - o_j{}^2)$$

$$= \eta \cdot (t_j - o_j) \cdot (1 - o_j{}^2)$$

Now, the training rule for the hidden unit weights:

$$\frac{\vartheta E_d}{\vartheta net_j} = \sum_{k \in downstream(j)} \frac{\vartheta E_d}{\vartheta net_k} \cdot \frac{\vartheta net_k}{\vartheta net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \cdot \frac{\vartheta net_k}{\vartheta net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \cdot \frac{\vartheta net_k}{O_j} \cdot \frac{O_j}{\vartheta net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \cdot w_{kj} \cdot \frac{O_j}{\vartheta net_j}$$

$$= \sum_{k \in downstream(j)} -\delta_k \cdot w_{kj} \cdot (1 - o_j{}^2)$$

$$\delta_j = (1 - o_j{}^2) \cdot \sum_{k \in downstream(j)} \delta_k \cdot w_{kj}$$

Therefore,

$$\Delta w_{ji} = \eta \cdot \delta_j \cdot x_{ji}$$

$$\Delta w_{ji} = \eta \cdot x_{ji} \cdot (1 - o_j{}^2) \cdot \sum_{k \in downstream(j)} \delta_k \cdot w_{kj}$$

3. Training a Neural Network with Keras:

a) A printout of the part of the code changed:

```
# course: TCSS455
# ML in Python, homework 3
# date: 13/05/2019
# name: Martine De Cock
# description: Neural network for predicting personality of Facebook users

from keras.models import Sequential
from keras.layers import Dense
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn import metrics
import numpy as np

# Fix random seed for reproducibility
seed = 7
np.random.seed(seed)

# Loading the data
# There are 9500 users (rows)
# There are 81 columns for the LIWC features followed by columns for
# openness, conscientiousness, extraversion, agreeableness, neuroticism
# As the target variable, we select the extraversion column (column 83)
dataset = np.loadtxt("Facebook-User-LIWC-personality-HW3.csv", delimiter=",")
X = dataset[:,0:81]
y = dataset[:,83]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1500)

# Training and testing a linear regression model
linreg = LinearRegression()
linreg.fit(X_train,y_train)
y_pred = linreg.predict(X_test)
print('MSE with linear regression:', metrics.mean_squared_error(y_test, y_pred))

# Training and testing a neural network
model = Sequential()
model.add(Dense(3, input_dim=81, kernel_initializer='normal', activation='relu'))
model.add(Dense(2, kernel_initializer='normal', activation='relu'))
model.add(Dense(1, kernel_initializer='normal'))
model.compile(optimizer='adam', loss='mse', metrics=['mse'])
model.fit(X_train,y_train, epochs=100)
y_pred = model.predict(X_test)
print('MSE with neural network:', metrics.mean_squared_error(y_test, y_pred))
```

b) Screenshots of resulting MSE's:

```
MacBookdeMacBook-Pro:hw3_export Tianyi$ python3 hw3.py
Using TensorFlow backend.
MSE with linear regression: 0.6425651258434265
Epoch 1/100
8000/8000 [==============================] - 0s 48us/step - loss: 11.9516 - mean_squared_error: 11.9516
Epoch 2/100
8000/8000 [==============================] - 0s 25us/step - loss: 1.6126 - mean_squared_error: 1.6126
Epoch 3/100
8000/8000 [==============================] - 0s 24us/step - loss: 1.0749 - mean_squared_error: 1.0749
Epoch 4/100
8000/8000 [==============================] - 0s 30us/step - loss: 0.8757 - mean_squared_error: 0.8757
```

...

```
8000/8000 [==============================] - 0s 24us/step - loss: 0.6426 - mean_squared_error: 0.6426
Epoch 93/100
8000/8000 [==============================] - 0s 25us/step - loss: 0.6309 - mean_squared_error: 0.6309
Epoch 94/100
8000/8000 [==============================] - 0s 24us/step - loss: 0.7428 - mean_squared_error: 0.7428
Epoch 95/100
8000/8000 [==============================] - 0s 24us/step - loss: 0.6345 - mean_squared_error: 0.6345
Epoch 96/100
8000/8000 [==============================] - 0s 27us/step - loss: 0.6328 - mean_squared_error: 0.6328
Epoch 97/100
8000/8000 [==============================] - 0s 25us/step - loss: 0.6299 - mean_squared_error: 0.6299
Epoch 98/100
8000/8000 [==============================] - 0s 24us/step - loss: 0.6280 - mean_squared_error: 0.6280
Epoch 99/100
8000/8000 [==============================] - 0s 25us/step - loss: 0.6287 - mean_squared_error: 0.6287
Epoch 100/100
8000/8000 [==============================] - 0s 24us/step - loss: 0.6275 - mean_squared_error: 0.6275
MSE with neural network: 0.6365945184768219
```

c) A brief description of interesting aspects about training neural networks:

I always know that the number of epochs matters a lot, but I thought it would be more the better MSE results the code will yield. It turns out that is not the case. Since MSEs and losses are conflicted: as one goes up, the other one goes down. Losses go up and down along with different numbers epochs likes curves, which leads to the variation of MSEs. Therefore, choosing a right number epochs matter.

d) Electronic submission: *Please see on Canvas.*