CSC411 Project 3 bonus

March 23, 2018

Christopher Sheedy 1002362542, Shahin Jafari 1002401634

Contents

Part 1, a)

In project 3 we explored almost every algorithm learned in class, except neural networks. We decided to use neural networks for this bonus part, mainly due to the fact that it wasn't explored in project 3, and its flexibility in terms of tuning and hyper parameters.

The data used in project 3, was coming from a single Australian source (as mentioned in the handout) and it was cleaned to have certain criteria, such as restricting the real headlines to be after October 2016 and have the word 'Trump'. The split of the real and fake data was also uneven, meaning that 60% of the data was real and only 40% of it was fake. As a consequence, our classifier was better in terms of predicting real headlines.

We extracted all the data from the Kaggle dataset to be consisted with project 3, and we used the cleaning script, but removed the proposed criteria to include all the data. One issue that we encountered in this part was again the uneven ratio of the fake news and real news. To remove this issue, we limited the number of real headlines to be equal to the fake headlines. Now having two sets of data, we can summarize their information:

Table 1: Summary of the data used

	Project 3 data	Entire Kaggle Data
Size of the training set	2286	11834
Size of the validation set	491	2536
Size of the test set	489	2534
Vocabulary vector size	5832 x 1	20478 x 1

In order to be able to compare the result of the neural network vs. the other methods explored, we first used the project 3 data. In defining the neural network hidden layers and number of neurons, an exhaustive search was conducted. The first layer (input) size was 5238, which is the same as the vocabulary size, and the output was kept as a sigmoid function. We allowed the number of hidden layers to range from 1 to 5, with Tanh activation. The number of neurons in each hidden layer was set to start with 512, and decrease by a factor of 2 in each hidden layer. That is, if we define our network to have 1 hidden layer, then that layer would have 512 neurons, where as if it was defined to have 3 layers, layer 1 would have 512 neurons; layer 2 would be consisted of 256 neurons and layer 3 would include 128 neurons, all with Tanh activation. The learning rate was searched from 1×10^{-2} to 1×10^{-4}. The maximum iteration had the freedom of being one of the [100,250,500,750,1000]. The performance of the network was evaluated against the validation set. The results were as follows:

Table 2: Neural Network Hyperparameter search

Iterations	Learning Rate	Hidden Layers	Performance on Validation Set
100	0.01	1	0.810590631
100	0.01	2	0.796334012
100	0.01	3	0.763747454
100	0.01	4	0.800407332
100	0.01	5	0.802443992
100	0.001	1	0.820773931
100	0.001	2	0.814663951
100	0.001	3	0.800407332
100	0.001	4	0.802443992
100	0.001	5	0.804480652
100	0.0001	1	0.820773931
100	0.0001	2	0.847250509
100	0.0001	3	0.843177189
100	0.0001	4	0.843177189
100	0.0001	5	0.845213849
250	0.01	1	0.810590631
250	0.01	2	0.796334012
250	0.01	3	0.763747454
250	0.01	4	0.782077393
250	0.01	5	0.800407332
250	0.001	1	0.810590631
250	0.001	2	0.806517312
250	0.001	3	0.802443992
250	0.001	4	0.804480652
250	0.001	5	0.798370672
250	0.0001	1	0.83706721
250	0.0001	2	0.818737271
250	0.0001	3	0.822810591
250	0.0001	4	0.820773931
250	0.0001	5	0.822810591
500	0.01	1	0.810590631
500	0.01	2	0.796334012
500	0.01	3	0.763747454
500	0.01	4	0.778004073
500	0.01	5	0.800407332
500	0.001	1	0.808553971
500	0.001	2	0.806517312
500	0.001	3	0.802443992
500	0.001	4	0.804480652
500	0.001	5	0.800407332
500	0.0001	1	0.82688391
500	0.0001	2	0.816700611
500	0.0001	3	0.812627291
500	0.0001	4	0.816700611
500	0.0001	5	0.820773931

Table 3: Neural Network Hyperparameter search

Iterations	Learning Rate	Hidden Layers	Performance on Validation Set
750	0.01	1	0.810590631
750	0.01	2	0.796334012
750	0.01	3	0.763747454
750	0.01	4	0.775967413
750	0.01	5	0.784114053
750	0.001	1	0.806517312
750	0.001	2	0.800407332
750	0.001	3	0.804480652
750	0.001	4	0.802443992
750	0.001	5	0.788187373
750	0.0001	1	0.824847251
750	0.0001	2	0.812627291
750	0.0001	3	0.808553971
750	0.0001	4	0.808553971
750	0.0001	5	0.820773931
1000	0.01	1	0.812627291
1000	0.01	2	0.796334012
1000	0.01	3	0.761710794
1000	0.01	4	0.773930754
1000	0.01	5	0.735234216
1000	0.001	1	0.802443992
1000	0.001	2	0.796334012
1000	0.001	3	0.806517312
1000	0.001	4	0.802443992
1000	0.001	5	0.788187373
1000	0.0001	1	0.820773931
1000	0.0001	2	0.810590631
1000	0.0001	3	0.810590631
1000	0.0001	4	0.810590631
1000	0.0001	5	0.820773931

From the tables above, we chose the number of hidden layers to be 2, with the learning rate of 0.0001, and maximum iterations of 100, resulting in a performance on the validation set of 84.73%. The table below will shows the summary of all the explored models and their performances:

Table 4: Summary of performances

	Training Set Accuracy	Validation Set Accuracy	Test Set Accuracy	Difference Between Training and Validation
Naïve Bayes	93.1	79.8	77.3	13.3
Logistic Regression	82.59	78	79.35	4.59
Decision Tree	97.94	75.97	74.44	21.97
Neural Network	98.51	84.73	85.28	13.78

Again, we can conclude that the neural network approach to this classification has the highest accuracy on the test set and it is about 6% better than the best method in project 3 which was the logistic regression approach. It does a bit of overfitting, but it does not compromise the test set performance.

We then moved to try our best model obtained from the project 3 dataset on the larger dataset (Kaggle) to observe the performance, and conclude if the model overfitted some words specific to the headlines in project 3 dataset. The learning curves are presented below:

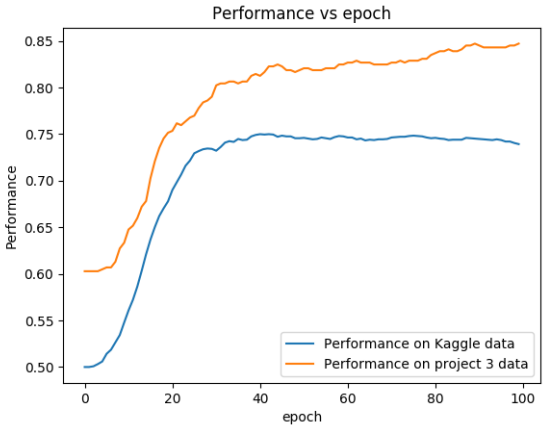

Figure 1: Learning Curve. Performance vs. Number of Iterations

From the result of the search, we knew that it would plateau, so we did not generate more epochs to save on computation time. The model did well in terms of predicting news in the entire Kaggle dataset. Part of the reason is due to the fact the project 3 dataset is a subset of the Kaggle dataset, and the model was at least capable of predicting the intersection of the datasets, and a maybe a little bit more.

We tried training the same model with the entire data and observe the performance on the dataset provided in project 3. Below are the learning curves:

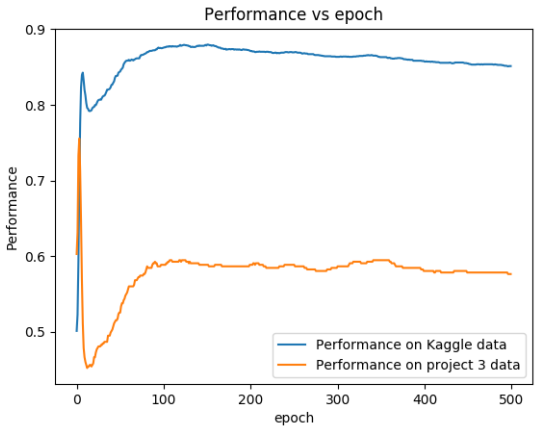

Figure 2: Learning Curve. Performance vs. Number of Iterations

As can be seen from the learning curve, this time the model was not capable of predicting the project 3 dataset properly. The reason could be due the fact the project 3 dataset is again a subset of the Kaggle data. The respective set difference of the project 3 data and the Kaggle data is large, and has a lot of more keywords (20 thousands vs 5 thousands) and there are lots of other news in the Kaggle data that can have common keywords between themselves, that are not keywords in the project 3 dataset. As a result, the neural network will associate higher weights to those specific keywords when trained on the Kaggle dataset, and the performance of it will be high for the Kaggle dataset. But because those keywords were not represented or were not major keywords in the project 3 dataset, the performance on the project 3 dataset is low.

Part 1, b)

To see how the neural net in part a classifies the news, we looked at the derivative of the output layer with respect to each keyword, which was represented as an input. The sign of the weight is a representative of a keyword making the prediction lean toward a class, mainly real news (positive derivative) and fake news (negative derivative). By comparing their respective magnitudes, we can find out which keywords influence our prediction the most, and we can compare them with the keywords obtained in project 3, with Naive Bayes method and Logistic regression method. To do this, we made artificial headlines that include a single word. We passed them through the network and computed their gradient. Below are the most influential keywords, with highest positive and negative derivative, and a summary of the keywords found in project 3:

Table 5: top ten predictors for real news

Method	words
Naive Bayes	'real', 'korea', 'turnbull', 'travel', 'australia', 'paris', 'tax', 'comments', 'trumps', 'refugee', 'congress'
Logistic Regression	'trump', new', 'donald', 'election', 'media', 'executive', 'comments', 'calls', 'korea', 'gets'
Neural Network	'wearing', 'half', 'desire', 'leak', 'stories', 'behold', 'tapping', 'contestants', 'really', 'glory'

Table 6: top ten predictors for fake news

Method	words
Naive Bayes	'trump', 'hillary', 'just', 'donald', 'clinton', 'comment', 'watch', 'new', 'win', 'breaking'
Logistic Regression	'remember', 'court', 'masks', 'concedes', 'open', 'trumps', 'beginning', 'load', 'clinton', 'year'
Neural Network	'expect', 'tests', 'means', 'issues', 'spies', 'envoy', 'entire', 'russian', 'yearns', 'growth'

With this approach, we did not get the same results as the Naive Bayes model or the logistic regression model, but the keywords are not very odd. In particular, keywords extracted from other models indeed have the expected sign in the neural network derivative. For example, the keyword trump has a positive sign which means it is an indicator of the real news, and we can see that trump appears in the top predictors of the real news in logistic regression. Indeed all the top keywords in the logistic regression model, had the correct sign in the neural network derivative. The reason that they did not appear as the top keywords in neural network was their respective magnitude. Interestingly, the same logic also applies backwards, meaning that the keywords with positive derivative in nerual network have a positive weight in logistic regression model, and those with negative derivatives, are represented by a negative weight in logistic regression.

Part 2)

Unfortunately, we decided not to pursue this part of the bonus assignment.

Code

```
import numpy as np
from sklearn.feature_extraction.text import ENGLISH_STOP_WORDS
import matplotlib.pyplot as plt
import matplotlib.cbook as cbook
import random
import time
from scipy.misc import imread
from scipy.misc import imresize
import matplotlib.image as mpimg
import sklearn.tree as tree
import graphviz


import torch
from torch.autograd import Variable
import urllib
import hashlib
import timeit
import random

random.seed(0)

def get_test_train_valid():
    np.random.seed(0)
    all_real = np.array([])
    all_fake = np.array([])

    for line in open("clean_real.txt"):
        all_real = np.append(all_real, line)

    for line in open("clean_fake.txt"):
        all_fake = np.append(all_fake, line)

    #shuffle sets
    np.random.shuffle(all_real)
    np.random.shuffle(all_fake)

    #split sets
    real_test_index = int(np.floor(0.15*np.size(all_real)))
    real_train_index = int(np.floor(0.85*np.size(all_real)))
    real_test, real_train, real_valid = np.split(all_real,[real_test_index, real_train_index])

    fake_test_index = int(np.floor(0.15*np.size(all_fake)))
    fake_train_index = int(np.floor(0.85*np.size(all_fake)))
```

```
    fake_test, fake_train, fake_valid = np.split(all_fake, [fake_test_index, fake_train_index])

    return real_test, real_train, real_valid, fake_test, fake_train, fake_valid

def get_test_train_valid_more():
    np.random.seed(0)
    all_real = np.array([])
    all_fake = np.array([])


    for line in open("clean_fake_new.txt"):
        all_fake = np.append(all_fake, line)

    cnt = 0
    for line in open("clean_real_new.txt"):
        if(cnt == len(all_fake)):
            break
        cnt += 1
        all_real = np.append(all_real, line)

    #shuffle sets
    np.random.shuffle(all_real)
    np.random.shuffle(all_fake)

    #split sets
    real_test_index = int(np.floor(0.15*np.size(all_real)))
    real_train_index = int(np.floor(0.85*np.size(all_real)))
    real_test, real_train, real_valid = np.split(all_real,[real_test_index, real_train_index])

    fake_test_index = int(np.floor(0.15*np.size(all_fake)))
    fake_train_index = int(np.floor(0.85*np.size(all_fake)))
    fake_test, fake_train, fake_valid = np.split(all_fake, [fake_test_index, fake_train_index])

    return real_test, real_train, real_valid, fake_test, fake_train, fake_valid

def get_vocab_og():

    vocab = []
    map = {}

    for line in open("clean_fake.txt"):
        words = str.split(line)
        for word in words:
            if word not in vocab:
                vocab.append(word)
```

```
    for line in open("clean_real.txt"):
        words = str.split(line)
        for word in words:
            if word not in vocab:
                vocab.append(word)

    for word in vocab:
        map[word] = vocab.index(word)

    return map

def get_logistic_sets(real_test, real_train, real_valid, fake_test, fake_train, fake_valid,vocab =
    ↪ get_vocab_og):

    vocab = vocab()
    x = np.zeros((len(vocab),real_train.shape[0] + fake_train.shape[0]))
    y = np.zeros((1,real_train.shape[0] + fake_train.shape[0]))

    case = 0
    for news in real_train:
        words = str.split(news)
        for word in words:
            if word not in vocab:
                continue
            row = vocab.get(word)
            x[row,case] = 1
        y[0,case] = 1
        case += 1

    for news in fake_train:
        words = str.split(news)
        for word in words:
            if word not in vocab:
                continue
            row = vocab.get(word)
            x[row,case] = 1
        y[0,case] = 0
        case += 1

    train_set = x
    train_set_label = y

    x = np.zeros((len(vocab),real_test.shape[0] + fake_test.shape[0]))
    y = np.zeros((1,real_test.shape[0] + fake_test.shape[0]))

    case = 0
```

```
for news in real_test:
    words = str.split(news)
    for word in words:
        if word not in vocab:
            continue
        row = vocab.get(word)
        x[row,case] = 1
    y[0,case] = 1
    case += 1

for news in fake_test:
    words = str.split(news)
    for word in words:
        if word not in vocab:
            continue
        row = vocab.get(word)
        x[row,case] = 1
    y[0,case] = 0
    case += 1

test_set = x
test_set_label = y


x = np.zeros((len(vocab),real_valid.shape[0] + fake_valid.shape[0]))
y = np.zeros((1,real_valid.shape[0] + fake_valid.shape[0]))

case = 0
for news in real_valid:
    words = str.split(news)
    for word in words:
        if word not in vocab:
            continue
        row = vocab.get(word)
        x[row,case] = 1
    y[0,case] = 1
    case += 1

for news in fake_valid:
    words = str.split(news)
    for word in words:
        if word not in vocab:
            continue
        row = vocab.get(word)
        x[row,case] = 1
    y[0,case] = 0
```

```
        case += 1

    valid_set = x
    valid_set_label = y

    return train_set,train_set_label, test_set, test_set_label, valid_set, valid_set_label

def get_vocab():

    vocab = []
    map = {}

    cnt_fake = 0
    for line in open("clean_fake_new.txt"):
        cnt_fake += 1
        words = str.split(line)
        for word in words:
            if word not in vocab:
                vocab.append(word)

    cnt = 0
    for line in open("clean_real_new.txt"):
        if (cnt == cnt_fake):
            break
        cnt += 1
        words = str.split(line)
        for word in words:
            if word not in vocab:
                vocab.append(word)

    for word in vocab:
        map[word] = vocab.index(word)

    return mapx`x'
def bonus_nn_torch():

    np.random.seed(0)
    torch.random.manual_seed(0)

    #vocabulary from project 3 data
    real_test, real_train, real_valid, fake_test, fake_train, fake_valid = get_test_train_valid()
    train_set,train_set_label, test_set, test_set_label, valid_set, valid_set_label = get_logistic_sets(
        ↪ real_test, real_train, real_valid, fake_test, fake_train, fake_valid,get_vocab_og)
    real_test_more, real_train_more, real_valid_more, fake_test_more, fake_train_more,
        ↪ fake_valid_more = get_test_train_valid_more()
```

```
train_set_more,train_set_label_more, test_set_more, test_set_label_more, valid_set_more,
    ↪ valid_set_label_more = get_logistic_sets(real_test_more, real_train_more,
    ↪ real_valid_more, fake_test_more, fake_train_more, fake_valid_more,get_vocab_og)

#vocabulary from kaggle data
# real_test, real_train, real_valid, fake_test, fake_train, fake_valid = get_test_train_valid()
# train_set,train_set_label, test_set, test_set_label, valid_set, valid_set_label = get_logistic_sets(
    ↪ real_test, real_train, real_valid, fake_test, fake_train, fake_valid,get_vocab)
# real_test_more, real_train_more, real_valid_more, fake_test_more, fake_train_more,
    ↪ fake_valid_more = get_test_train_valid_more()
# train_set_more,train_set_label_more, test_set_more, test_set_label_more, valid_set_more,
    ↪ valid_set_label_more = get_logistic_sets(real_test_more, real_train_more,
    ↪ real_valid_more, fake_test_more, fake_train_more, fake_valid_more,get_vocab)

print('done loading sets')

dtype_float = torch.FloatTensor
dtype_long = torch.LongTensor
dim_x = train_set.shape[0]
dim_h1 = 512
dim_h2 = 256
dim_o = 1
max_iter = 100
learning_rate = 1e-4;

#train on small data
# x = Variable(torch.from_numpy(train_set.T), requires_grad = False).type(dtype_float)
# y = Variable(torch.from_numpy(train_set_label.T), requires_grad = False).type(dtype_float)

#train on kaggle data
x = Variable(torch.from_numpy(train_set_more.T), requires_grad = False).type(dtype_float)
y = Variable(torch.from_numpy(train_set_label_more.T), requires_grad = False).type(
    ↪ dtype_float)


model = torch.nn.Sequential( torch.nn.Linear(dim_x,dim_h1), torch.nn.Tanh(), torch.nn.
    ↪ Linear(dim_h1,dim_h2), torch.nn.Tanh(), torch.nn.Linear(dim_h2,dim_o), torch.nn.
    ↪ Sigmoid())
loss_fn = torch.nn.BCELoss()

optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)


performance = np.zeros((max_iter,3))

for t in range(0,max_iter):
```

```
        y_pred = model(x)
        loss = loss_fn(y_pred, y)

        model.zero_grad()
        loss.backward()
        optimizer.step()

        x_perf = Variable(torch.from_numpy(valid_set_more.T), requires_grad = False).type(
            ↪ dtype_float)
        y_pred_test = model(x_perf).data.numpy()
        score_more = np.mean(np.round(y_pred_test[:,0]) == valid_set_label_more.reshape(
            ↪ valid_set_label_more.shape[1],))

        x_perf = Variable(torch.from_numpy(valid_set.T), requires_grad = False).type(dtype_float
            ↪ )
        y_pred_test = model(x_perf).data.numpy()
        score_less = np.mean(np.round(y_pred_test[:,0]) == valid_set_label.reshape(valid_set_label
            ↪ .shape[1],))

        performance[t,0] = t
        performance[t,1] = score_more
        performance[t,2] = score_less
        print('iter=' + str(t) + ' more:' + str(score_more) + ' less:' + str(score_less))

    #plot the learning curve
    plt.figure()
    plt.plot(performance[:,0],performance[:,1]) #train set plot
    plt.plot(performance[:,0],performance[:,2]) #valid set plot
    plt.title('Performance vs epoch')
    plt.xlabel('epoch')
    plt.ylabel('Performance')
    plt.legend(['Performance on Kaggle data', 'Performance on project 3 data'])
    plt.show()

    return model, performance


def bonus_nn_search():

    np.random.seed(0)
    torch.random.manual_seed(0)

    real_test, real_train, real_valid, fake_test, fake_train, fake_valid = get_test_train_valid()

    train_set,train_set_label, test_set, test_set_label, valid_set, valid_set_label = get_logistic_sets(
        ↪ real_test, real_train, real_valid, fake_test, fake_train, fake_valid,get_vocab_og)
```

```
print('done loading sets')

iterations = [100,250,500,750,1000]
alphas = [1e-2,1e-3,1e-4]
hidden = [1,2,3,4,5]
neurons = [512,256,128,64,32]

dtype_float = torch.FloatTensor
dtype_long = torch.LongTensor
dim_x = train_set.shape[0]

x = Variable(torch.from_numpy(train_set.T), requires_grad = False).type(dtype_float)
y = Variable(torch.from_numpy(train_set_label.T), requires_grad = False).type(dtype_float)
performance = np.zeros((len(iterations)*len(hidden)*len(alphas),4))
i = 0

for max_iter in iterations:
    for learning_rate in alphas:
        for layer in hidden:
            torch.random.manual_seed(0)
            model = torch.nn.Sequential()
            loss_fn = torch.nn.BCELoss()
            model.add_module('input',torch.nn.Linear(dim_x,neurons[0]))
            for layer_cnt in range (1,layer+1):
                model.add_module('activation'+str(layer_cnt), torch.nn.Tanh())
                if (layer_cnt+1) in range(1,layer+1):
                    output_dim = neurons[layer_cnt]
                else:
                    output_dim = 1
                model.add_module('fc'+str(layer_cnt),torch.nn.Linear(neurons[layer_cnt-1],
                    ↪ output_dim))
            model.add_module('output', torch.nn.Sigmoid())
            optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
            for t in range(0,max_iter):
                y_pred = model(x)
                loss = loss_fn(y_pred, y)

                model.zero_grad()
                loss.backward()
                optimizer.step()

            x_perf = Variable(torch.from_numpy(valid_set.T), requires_grad = False).type(
                ↪ dtype_float)
            y_pred_test = model(x_perf).data.numpy()
```

```
                score = np.mean(np.round(y_pred_test[:,0]) == valid_set_label.reshape(
                    ↪ valid_set_label.shape[1],))

                performance[i,0] = max_iter
                performance[i,1] = learning_rate
                performance[i,2] = layer
                performance[i,3] = score

                print('max_iter:' + str(max_iter) + ' layers:' + str(layer) + ' learning_rate:' + str
                    ↪ (learning_rate) + ' performance:' + str(score))
                i += 1


    return performance


def bonus_nn_derivatives():

    #parameters are obtained from search
    np.random.seed(0)
    torch.random.manual_seed(0)

    real_test, real_train, real_valid, fake_test, fake_train, fake_valid = get_test_train_valid()

    train_set,train_set_label, test_set, test_set_label, valid_set, valid_set_label = get_logistic_sets(
        ↪ real_test, real_train, real_valid, fake_test, fake_train, fake_valid,get_vocab_og)

    print('done loading sets')

    dtype_float = torch.FloatTensor
    dtype_long = torch.LongTensor
    dim_x = train_set.shape[0]
    dim_h1 = 512
    dim_h2 = 256
    dim_o = 1
    max_iter = 100
    learning_rate = 1e-4;

    x = Variable(torch.from_numpy(train_set.T), requires_grad = False).type(dtype_float)
    y = Variable(torch.from_numpy(train_set_label.T), requires_grad = False).type(dtype_float)


    model = torch.nn.Sequential( torch.nn.Linear(dim_x,dim_h1), torch.nn.Tanh(), torch.nn.
        ↪ Linear(dim_h1,dim_h2), torch.nn.Tanh(), torch.nn.Linear(dim_h2,dim_o), torch.nn.
        ↪ Sigmoid())
    loss_fn = torch.nn.BCELoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)


for t in range(0,max_iter):
    y_pred = model(x)
    loss = loss_fn(y_pred, y)

    model.zero_grad()
    loss.backward()
    optimizer.step()

print('done training')
derivatives = np.zeros((train_set.shape[0],1))
for i in range(0,train_set.shape[0]):
    headline = np.zeros((train_set.shape[0],1))
    headline[i,0] = 1
    x = Variable(torch.from_numpy(headline.T), requires_grad = True).type(dtype_float)
    derivatives[i,0] = torch.autograd.grad(model(x),x)[0][0,i].data[0]


return derivatives
```