Protein Family Classification using Recurrent Neural Networks and Distributed Representations

Data Book

Tejaswi Krishna Vemulapati

Installation of TensorFlow

February 4, 2018

After a lot of trial and error (dealing with Python package version incompatibilities), I have found that the following is the best and easiest way to install the TensorFlow python package on my mac os computer. This method is known as VirtualEnv installation because VirtualEnv is a tool which keeps the dependencies required by different python projects in separate virtual environments.
The following are the steps to install TensorFlow using the VirtualEnv method.

1. Install pip which is package management software for packages written in python.

   ```
   sudo easy_install pip
   ```

   This install pip version 9.0.1.
2. Install VirtualEnv.

   ```
   sudo pip install --upgrade virtualenv
   ```

3. Create a virtual environment in the directory ~/virtualenv.

   ```
   virtualenv --system-site-packages ~/tensorflow
   ```

4. Activate the virtualenv environment.

   ```
   source ~/tensorflow/bin/activate  # If using bash
   ```

   The command line prompt changes to:

   ```
   (tensorflow) $
   ```

5. Install TensorFlow in the newly created virtual environment. First choose the correct binary of tensoeflow to install. I have chosen the TensorFlow binary for Mac OS X which is CPU-only (that is no GPU support) for python 2.7.

   ```
   export
   TF_BINARY_URL=https://storage.googleapis.com/tensorflow/mac/cpu/tensorflow-
   0.12.1-py2-none-any.whl
   pip install --upgrade $TF_BINARY_URL
   ```

6. Test the TensorFlow installation.
   The following is a simple program using the TensorFlow package.

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
a = tf.constant(10)
b = tf.constant(32)
print(sess.run(a + b))
```

The following is the output.

```
$ python ~/predict/hello.py
  Hello, TensorFlow!
  42
```

7. After done with using TensorFlow, deactivate the environmenent:

```
$ deactivate
```

Simple Matrix Multiplication in TensorFlow

February 05, 2018

Today, my goal is to get familiar with TensorFlow by writing a simple program to multiply two matrices in TensorFlow.

First, some terminology.

What is a tensor?

A tensor is a multidimenstional array. Some special cases are:

- scalars like numbers 1, 3.1415 etc. are rank zero tensors.
- Vectors like the three dimenstional vector [2, 4, 10] are rank one tensors.
- Matrices such as $\begin{bmatrix} 2 & 4 \\ 8 & 6 \end{bmatrix}$ are rank two tensors.

In TensorFlow all data are represented as tensors.

Any TensorFlow program has two basic parts. In the first part, we need to build a computation graph. In the second part, the computation graph is executed in the context of a TensorFlow session. A computation graph consists of operators and operands. Each operator takes zero or more input operands and executes the specified operation on those operands and outputs one or more operands. For example, in the case of matrix multiplication, the operator **matmul** of TensorFlow takes two constant operands (which are the input matrices to be multiplied) and outputs another matrix which is the product of those matrices. In my example the matrix multiplication is as follows:

$$\begin{bmatrix} 2.0 & 3.0 \\ 1.0 & 4.0 \\ 8.0 & 2.0 \end{bmatrix} X \begin{bmatrix} 1.0 & 3.0 & 2.0 \\ 6.0 & 1.0 & 4.0 \end{bmatrix} = \begin{bmatrix} 20.0 & 9.0 & 16.0 \\ 25.0 & 7.0 & 18.0 \\ 20.0 & 26.0 & 24.0 \end{bmatrix}$$

The following is the TensorFlow program to perform the matrix multiplication.

- Note that matrices are represented row major. That means, the columns in a row are input before moving to the next row.
- Constants are operators with zero input operands.
- The run method of TensorFlow session is invoked with an operand which is the final node in the computation graph. In our example, this is the result of the matrix multiplication.

```
# matrix multiplication in tensorflow
import tensorflow as tf

# matrix1 is 3x2
matrix1 = tf.constant([[2.0, 3.0], [1.0, 4.0], [8.0, 2.0]])

# matrix1 is 2x3
matrix2 = tf.constant([[1.0, 3.0, 2.0], [6.0, 1.0, 4.0]])

# matrix1 x matrix2 is 3x3

matrix3  = tf.matmul(matrix1, matrix2)

# we have just created the computation graph. We need
# to execute the graph in a session.

with tf.Session() as session:
    result = session.run(matrix3)
    print (result)
    # the above prints [[ 20.   9.  16.]
    #                  [ 25.   7.  18.]
    #                  [ 20.  26.  24.]]
```

Simple Linear Regression in TensorFlow
February 10, 2018

Today my goal is to implement a simple linear regression model in Tensorflow.

The code

```
'''
A linear regression learning algorithm example using TensorFlow library.
Author: Tejaswi Vemulapati
'''

from __future__ import print_function

import tensorflow as tf
import numpy
import matplotlib.pyplot as plt
rng = numpy.random

# Parameters
learning_rate = 0.001
training_epochs = 10240
display_step = 50

# slope (W): 2.34
# displacement(b): 3.94
# Training Data
train_X =
numpy.asarray([1.97,5.74,0.58,6.37,3.93,0.86,5.91,3.01,5.81,5.54,3.68,2.65,5.2,7.96,3.29,3.41,6.21,7.5
1,3.38,0.87,3.76,6.93,3.95,2.48,7.8,5.2,2.76,3.75,5.39,1.24,7.56,0.51])
train_Y =
numpy.asarray([8.6498,17.6316,5.2772,19.1958,13.6062,5.5724,17.3294,11.4334,17.3254,17.0136,12.
4212,9.691,15.968,22.9364,11.2786,12.2494,18.6314,21.6834,11.4892,5.6058,12.6984,20.1662,12.693
,9.8532,22.312,16.198,10.5884,12.585,16.1026,6.3916,21.6504,5.3934])
n_samples = train_X.shape[0]

# tf Graph Input
X = tf.placeholder("float")
Y = tf.placeholder("float")

# Set model weights
W = tf.Variable(rng.randn(), name="slope")
b = tf.Variable(rng.randn(), name="displacement")

# Construct a linear model
pred = tf.add(tf.multiply(X, W), b)

# Mean squared error
cost = tf.reduce_sum(tf.pow(pred-Y, 2))/(2*n_samples)
```

```
# Gradient descent
#  Note, minimize() knows to modify W and b because Variable objects are trainable=True by default
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:
    writer = tf.summary.FileWriter('/Users/muralivemulapati/prot/graphs', sess.graph)
    # Run the initializer
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        for (x, y) in zip(train_X, train_Y):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(c), \
                "W=", sess.run(W), "b=", sess.run(b))

    print("Optimization Finished!")
    training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
    print("Training cost=", training_cost, "W=", sess.run(W), "b=", sess.run(b), '\n')

    # Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()

    # Testing example, as requested (Issue #2)
    test_X = numpy.asarray([6.83, 4.668, 8.9, 7.91, 5.7, 8.7, 3.1, 2.1])
    test_Y = numpy.asarray([1.84, 2.273, 3.2, 2.831, 2.92, 3.24, 1.35, 1.03])

    print("Testing... (Mean square loss Comparison)")
    testing_cost = sess.run(
        tf.reduce_sum(tf.pow(pred - Y, 2)) / (2 * test_X.shape[0]),
        feed_dict={X: test_X, Y: test_Y})  # same function as cost above
    print("Testing cost=", testing_cost)
    print("Absolute mean square loss difference:", abs(
        training_cost - testing_cost))

    plt.plot(test_X, test_Y, 'bo', label='Testing data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
```

```
    plt.show()
writer.close()
```

```
python linear_regression.py
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.1 instructions, but these are available on your machine and could speed up CPU computations.
W tensorflow/core/platform/cpu_feature_guard.cc:45] The TensorFlow library wasn't compiled to use SSE4.2 instructions, but these are available on your machine and could speed up CPU computations.
Epoch: 0050 cost= 18.683168411 W= 2.12296 b= -1.25084
Epoch: 0100 cost= 3.980549574 W= 2.91149 b= -1.05183
Epoch: 0150 cost= 2.525871992 W= 3.1494 b= -0.957061
Epoch: 0200 cost= 2.343295097 W= 3.21689 b= -0.894899
Epoch: 0250 cost= 2.283892155 W= 3.23166 b= -0.843167
Epoch: 0300 cost= 2.237251282 W= 3.23019 b= -0.794994
Epoch: 0350 cost= 2.192694902 W= 3.22374 b= -0.748251
Epoch: 0400 cost= 2.149158478 W= 3.21581 b= -0.702273
Epoch: 0450 cost= 2.106513500 W= 3.20748 b= -0.65685
Epoch: 0500 cost= 2.064737082 W= 3.19909 b= -0.611919
Epoch: 0550 cost= 2.023804665 W= 3.19074 b= -0.567452
Epoch: 0600 cost= 1.983700037 W= 3.18246 b= -0.523441
Epoch: 0650 cost= 1.944405079 W= 3.17426 b= -0.479876
Epoch: 0700 cost= 1.905905008 W= 3.16614 b= -0.436756
Epoch: 0750 cost= 1.868183255 W= 3.1581 b= -0.394072
Epoch: 0800 cost= 1.831224203 W= 3.15015 b= -0.351823
Epoch: 0850 cost= 1.795012474 W= 3.14228 b= -0.310003
Epoch: 0900 cost= 1.759532452 W= 3.13448 b= -0.268608
Epoch: 0950 cost= 1.724770069 W= 3.12677 b= -0.227633
Epoch: 1000 cost= 1.690709352 W= 3.11913 b= -0.187075
Epoch: 1050 cost= 1.657337308 W= 3.11157 b= -0.146928
Epoch: 1100 cost= 1.624639750 W= 3.10409 b= -0.107189
Epoch: 1150 cost= 1.592603922 W= 3.09669 b= -0.0678538
Epoch: 1200 cost= 1.561215162 W= 3.08935 b= -0.028918
Epoch: 1250 cost= 1.530461311 W= 3.0821 b= 0.009622
Epoch: 1300 cost= 1.500328660 W= 3.07492 b= 0.0477706
Epoch: 1350 cost= 1.470804095 W= 3.0678 b= 0.0855317
Epoch: 1400 cost= 1.441878319 W= 3.06077 b= 0.122909
Epoch: 1450 cost= 1.413535118 W= 3.0538 b= 0.159907
Epoch: 1500 cost= 1.385767579 W= 3.04691 b= 0.196529
Epoch: 1550 cost= 1.358558893 W= 3.04008 b= 0.232778
Epoch: 1600 cost= 1.331901550 W= 3.03333 b= 0.26866
Epoch: 1650 cost= 1.305781841 W= 3.02664 b= 0.304177
Epoch: 1700 cost= 1.280191183 W= 3.02002 b= 0.339333
Epoch: 1750 cost= 1.255117178 W= 3.01347 b= 0.374133
Epoch: 1800 cost= 1.230550528 W= 3.00699 b= 0.408578
Epoch: 1850 cost= 1.206479788 W= 3.00056 b= 0.442674
Epoch: 1900 cost= 1.182896614 W= 2.99421 b= 0.476423
Epoch: 1950 cost= 1.159789085 W= 2.98792 b= 0.509829
Epoch: 2000 cost= 1.137149453 W= 2.9817 b= 0.542897
Epoch: 2050 cost= 1.114966393 W= 2.97553 b= 0.575629
```

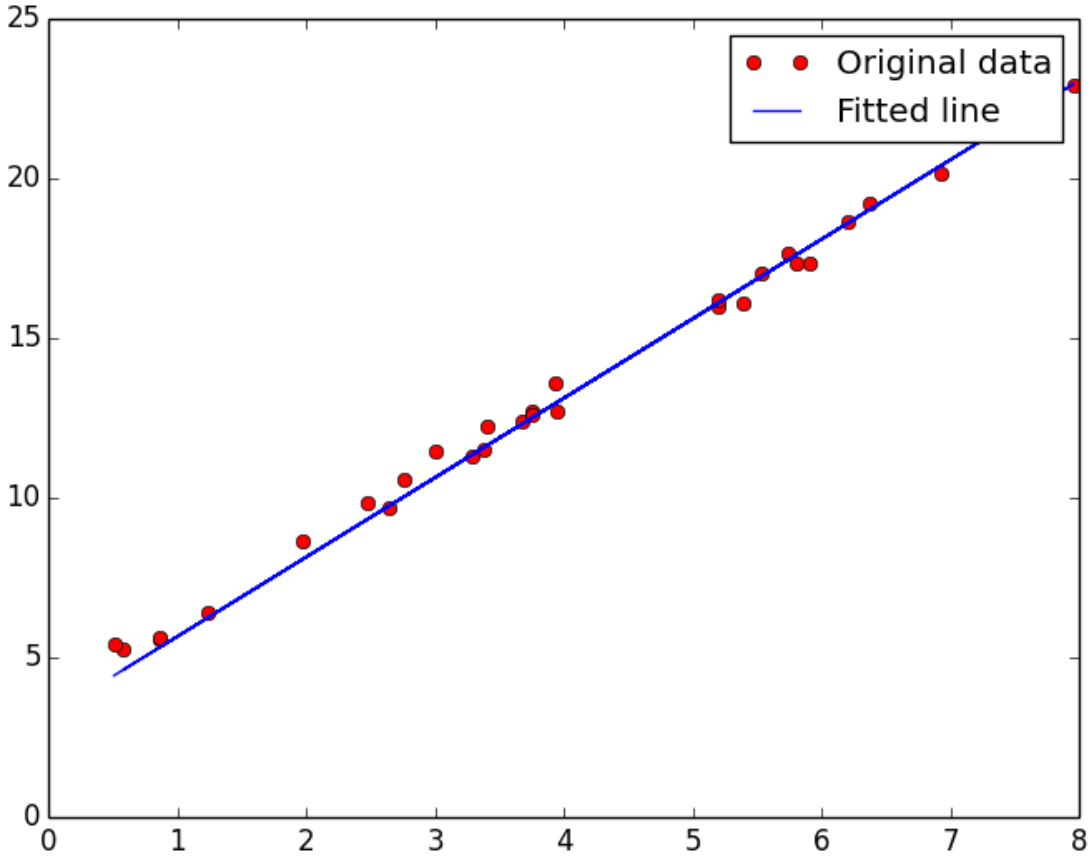
Epoch: 2100 cost= 1.093234539 W= 2.96943 b= 0.608025
Epoch: 2150 cost= 1.071938396 W= 2.96339 b= 0.640096
Epoch: 2200 cost= 1.051073790 W= 2.95742 b= 0.671841
Epoch: 2250 cost= 1.030633688 W= 2.9515 b= 0.70326
Epoch: 2300 cost= 1.010603905 W= 2.94564 b= 0.734362
Epoch: 2350 cost= 0.990978956 W= 2.93985 b= 0.76515
Epoch: 2400 cost= 0.971752524 W= 2.93411 b= 0.795621
Epoch: 2450 cost= 0.952913046 W= 2.92843 b= 0.825785
Epoch: 2500 cost= 0.934455276 W= 2.92281 b= 0.855642
Epoch: 2550 cost= 0.916370511 W= 2.91725 b= 0.885196
Epoch: 2600 cost= 0.898650289 W= 2.91174 b= 0.914451
Epoch: 2650 cost= 0.881288171 W= 2.90629 b= 0.943409
Epoch: 2700 cost= 0.864279628 W= 2.90089 b= 0.972069
Epoch: 2750 cost= 0.847612977 W= 2.89555 b= 1.00044
Epoch: 2800 cost= 0.831285119 W= 2.89026 b= 1.02852
Epoch: 2850 cost= 0.815282702 W= 2.88503 b= 1.05632
Epoch: 2900 cost= 0.799602747 W= 2.87984 b= 1.08384
Epoch: 2950 cost= 0.784245968 W= 2.87472 b= 1.11108
Epoch: 3000 cost= 0.769198895 W= 2.86965 b= 1.13803
Epoch: 3050 cost= 0.754452705 W= 2.86462 b= 1.16472
Epoch: 3100 cost= 0.740006924 W= 2.85964 b= 1.19113
Epoch: 3150 cost= 0.725852132 W= 2.85473 b= 1.21728
Epoch: 3200 cost= 0.711981297 W= 2.84985 b= 1.24317
Epoch: 3250 cost= 0.698393941 W= 2.84502 b= 1.26878
Epoch: 3300 cost= 0.685083389 W= 2.84025 b= 1.29413
Epoch: 3350 cost= 0.672039986 W= 2.83553 b= 1.31923
Epoch: 3400 cost= 0.659258485 W= 2.83085 b= 1.34408
Epoch: 3450 cost= 0.646736741 W= 2.82622 b= 1.36867
Epoch: 3500 cost= 0.634464979 W= 2.82163 b= 1.39302
Epoch: 3550 cost= 0.622441828 W= 2.8171 b= 1.41712
Epoch: 3600 cost= 0.610662460 W= 2.81261 b= 1.44097
Epoch: 3650 cost= 0.599123240 W= 2.80816 b= 1.46457
Epoch: 3700 cost= 0.587817967 W= 2.80376 b= 1.48794
Epoch: 3750 cost= 0.576741993 W= 2.79941 b= 1.51107
Epoch: 3800 cost= 0.565888584 W= 2.7951 b= 1.53396
Epoch: 3850 cost= 0.555255890 W= 2.79083 b= 1.55662
Epoch: 3900 cost= 0.544832230 W= 2.78661 b= 1.57906
Epoch: 3950 cost= 0.534620881 W= 2.78243 b= 1.60127
Epoch: 4000 cost= 0.524616122 W= 2.77829 b= 1.62325
Epoch: 4050 cost= 0.514815092 W= 2.77419 b= 1.64501
Epoch: 4100 cost= 0.505213559 W= 2.77013 b= 1.66654
Epoch: 4150 cost= 0.495805621 W= 2.76612 b= 1.68786
Epoch: 4200 cost= 0.486589342 W= 2.76215 b= 1.70895
Epoch: 4250 cost= 0.477558553 W= 2.75822 b= 1.72984
Epoch: 4300 cost= 0.468708873 W= 2.75432 b= 1.75051
Epoch: 4350 cost= 0.460036635 W= 2.75047 b= 1.77098
Epoch: 4400 cost= 0.451542437 W= 2.74666 b= 1.79123
Epoch: 4450 cost= 0.443219841 W= 2.74289 b= 1.81128
Epoch: 4500 cost= 0.435063034 W= 2.73914 b= 1.83113

Epoch: 4550 cost= 0.427071095 W= 2.73545 b= 1.85078
Epoch: 4600 cost= 0.419243068 W= 2.73179 b= 1.87022
Epoch: 4650 cost= 0.411573052 W= 2.72816 b= 1.88946
Epoch: 4700 cost= 0.404057562 W= 2.72458 b= 1.90852
Epoch: 4750 cost= 0.396693230 W= 2.72103 b= 1.92738
Epoch: 4800 cost= 0.389477313 W= 2.71751 b= 1.94605
Epoch: 4850 cost= 0.382407486 W= 2.71403 b= 1.96453
Epoch: 4900 cost= 0.375480771 W= 2.71059 b= 1.98282
Epoch: 4950 cost= 0.368696064 W= 2.70719 b= 2.00092
Epoch: 5000 cost= 0.362046540 W= 2.7038 b= 2.01884
Epoch: 5050 cost= 0.355528146 W= 2.70046 b= 2.03658
Epoch: 5100 cost= 0.349149078 W= 2.69716 b= 2.05413
Epoch: 5150 cost= 0.342894047 W= 2.69389 b= 2.07152
Epoch: 5200 cost= 0.336764842 W= 2.69065 b= 2.08872
Epoch: 5250 cost= 0.330764234 W= 2.68744 b= 2.10574
Epoch: 5300 cost= 0.324884951 W= 2.68427 b= 2.12259
Epoch: 5350 cost= 0.319122285 W= 2.68113 b= 2.13927
Epoch: 5400 cost= 0.313475639 W= 2.67803 b= 2.15579
Epoch: 5450 cost= 0.307940841 W= 2.67494 b= 2.17214
Epoch: 5500 cost= 0.302516699 W= 2.67189 b= 2.18833
Epoch: 5550 cost= 0.297206283 W= 2.66888 b= 2.20434
Epoch: 5600 cost= 0.292003810 W= 2.66589 b= 2.22019
Epoch: 5650 cost= 0.286906272 W= 2.66294 b= 2.23588
Epoch: 5700 cost= 0.281910509 W= 2.66002 b= 2.25141
Epoch: 5750 cost= 0.277011693 W= 2.65712 b= 2.2668
Epoch: 5800 cost= 0.272217184 W= 2.65426 b= 2.28201
Epoch: 5850 cost= 0.267519891 W= 2.65142 b= 2.29707
Epoch: 5900 cost= 0.262914032 W= 2.64861 b= 2.31199
Epoch: 5950 cost= 0.258403212 W= 2.64583 b= 2.32675
Epoch: 6000 cost= 0.253984421 W= 2.64308 b= 2.34136
Epoch: 6050 cost= 0.249653891 W= 2.64036 b= 2.35582
Epoch: 6100 cost= 0.245412648 W= 2.63766 b= 2.37013
Epoch: 6150 cost= 0.241258144 W= 2.63499 b= 2.38429
Epoch: 6200 cost= 0.237182438 W= 2.63235 b= 2.39832
Epoch: 6250 cost= 0.233192354 W= 2.62974 b= 2.41221
Epoch: 6300 cost= 0.229286790 W= 2.62716 b= 2.42593
Epoch: 6350 cost= 0.225458398 W= 2.6246 b= 2.43953
Epoch: 6400 cost= 0.221707076 W= 2.62207 b= 2.45298
Epoch: 6450 cost= 0.218031719 W= 2.61955 b= 2.4663
Epoch: 6500 cost= 0.214425951 W= 2.61707 b= 2.4795
Epoch: 6550 cost= 0.210894912 W= 2.61461 b= 2.49256
Epoch: 6600 cost= 0.207432464 W= 2.61218 b= 2.5055
Epoch: 6650 cost= 0.204047054 W= 2.60978 b= 2.51828
Epoch: 6700 cost= 0.200723767 W= 2.60739 b= 2.53095
Epoch: 6750 cost= 0.197469145 W= 2.60503 b= 2.54349
Epoch: 6800 cost= 0.194278464 W= 2.60269 b= 2.55591
Epoch: 6850 cost= 0.191153243 W= 2.60037 b= 2.5682
Epoch: 6900 cost= 0.188091904 W= 2.59808 b= 2.58036

Epoch: 6950 cost= 0.185093880 W= 2.59582 b= 2.59239
Epoch: 7000 cost= 0.182155877 W= 2.59357 b= 2.6043
Epoch: 7050 cost= 0.179279268 W= 2.59136 b= 2.61609
Epoch: 7100 cost= 0.176457122 W= 2.58916 b= 2.62777
Epoch: 7150 cost= 0.173693746 W= 2.58698 b= 2.63932
Epoch: 7200 cost= 0.170986161 W= 2.58483 b= 2.65075
Epoch: 7250 cost= 0.168329567 W= 2.5827 b= 2.66209
Epoch: 7300 cost= 0.165727645 W= 2.58058 b= 2.67331
Epoch: 7350 cost= 0.163176924 W= 2.57849 b= 2.68442
Epoch: 7400 cost= 0.160685256 W= 2.57643 b= 2.69538
Epoch: 7450 cost= 0.158239424 W= 2.57438 b= 2.70625
Epoch: 7500 cost= 0.155846104 W= 2.57236 b= 2.717
Epoch: 7550 cost= 0.153496996 W= 2.57036 b= 2.72766
Epoch: 7600 cost= 0.151198387 W= 2.56837 b= 2.73819
Epoch: 7650 cost= 0.148944706 W= 2.5664 b= 2.74862
Epoch: 7700 cost= 0.146736205 W= 2.56446 b= 2.75896
Epoch: 7750 cost= 0.144578159 W= 2.56254 b= 2.76915
Epoch: 7800 cost= 0.142459124 W= 2.56062 b= 2.77927
Epoch: 7850 cost= 0.140385538 W= 2.55874 b= 2.78927
Epoch: 7900 cost= 0.138353989 W= 2.55689 b= 2.79917
Epoch: 7950 cost= 0.136362806 W= 2.55504 b= 2.80897
Epoch: 8000 cost= 0.134413213 W= 2.55322 b= 2.81867
Epoch: 8050 cost= 0.132501021 W= 2.55141 b= 2.82828
Epoch: 8100 cost= 0.130626589 W= 2.54961 b= 2.83779
Epoch: 8150 cost= 0.128792211 W= 2.54784 b= 2.8472
Epoch: 8200 cost= 0.126992375 W= 2.54608 b= 2.85652
Epoch: 8250 cost= 0.125232100 W= 2.54435 b= 2.86573
Epoch: 8300 cost= 0.123504192 W= 2.54263 b= 2.87487
Epoch: 8350 cost= 0.121810466 W= 2.54093 b= 2.88391
Epoch: 8400 cost= 0.120155208 W= 2.53925 b= 2.89285
Epoch: 8450 cost= 0.118529394 W= 2.53758 b= 2.90171
Epoch: 8500 cost= 0.116938218 W= 2.53593 b= 2.91047
Epoch: 8550 cost= 0.115383081 W= 2.5343 b= 2.91913
Epoch: 8600 cost= 0.113852710 W= 2.53268 b= 2.92773
Epoch: 8650 cost= 0.112354666 W= 2.53107 b= 2.93623
Epoch: 8700 cost= 0.110888250 W= 2.5295 b= 2.94465
Epoch: 8750 cost= 0.109449923 W= 2.52793 b= 2.95298
Epoch: 8800 cost= 0.108039096 W= 2.52638 b= 2.96124
Epoch: 8850 cost= 0.106659658 W= 2.52483 b= 2.9694
Epoch: 8900 cost= 0.105305091 W= 2.52331 b= 2.9775
Epoch: 8950 cost= 0.103980750 W= 2.5218 b= 2.98549
Epoch: 9000 cost= 0.102684565 W= 2.52032 b= 2.9934
Epoch: 9050 cost= 0.101410776 W= 2.51883 b= 3.00125
Epoch: 9100 cost= 0.100163236 W= 2.51738 b= 3.00902
Epoch: 9150 cost= 0.098937854 W= 2.51593 b= 3.01672
Epoch: 9200 cost= 0.097738653 W= 2.5145 b= 3.02434
Epoch: 9250 cost= 0.096564755 W= 2.51308 b= 3.03188
Epoch: 9300 cost= 0.095415384 W= 2.51167 b= 3.03933
Epoch: 9350 cost= 0.094288312 W= 2.51028 b= 3.04672

Epoch: 9400 cost= 0.093182281 W= 2.5089 b= 3.05404
Epoch: 9450 cost= 0.092101038 W= 2.50753 b= 3.06127
Epoch: 9500 cost= 0.091040179 W= 2.50618 b= 3.06843
Epoch: 9550 cost= 0.090005219 W= 2.50487 b= 3.0755
Epoch: 9600 cost= 0.088986427 W= 2.50354 b= 3.08253
Epoch: 9650 cost= 0.087993503 W= 2.50224 b= 3.08944
Epoch: 9700 cost= 0.087020993 W= 2.50095 b= 3.09629
Epoch: 9750 cost= 0.086064473 W= 2.49967 b= 3.10309
Epoch: 9800 cost= 0.085128613 W= 2.4984 b= 3.10981
Epoch: 9850 cost= 0.084210977 W= 2.49714 b= 3.11647
Epoch: 9900 cost= 0.083311364 W= 2.4959 b= 3.12306
Epoch: 9950 cost= 0.082429856 W= 2.49467 b= 3.12959
Epoch: 10000 cost= 0.081567287 W= 2.49345 b= 3.13605
Epoch: 10050 cost= 0.080720767 W= 2.49226 b= 3.14245
Epoch: 10100 cost= 0.079893827 W= 2.49106 b= 3.14876
Epoch: 10150 cost= 0.079082854 W= 2.48989 b= 3.15502
Epoch: 10200 cost= 0.078289419 W= 2.48872 b= 3.1612
Optimization Finished!
Training cost= 0.0776664 W= 2.4878 b= 3.1661

Testing... (Mean square loss Comparison)
Testing cost= 137.474
Absolute mean square loss difference: 137.397
(tensorflow) murali-vemulapatis-Mac-Pro:prot muralivemulapati$


Original data
Fitted line
25
20
15
10
5
0
0
1
2
3
4
5
6
7
8

The engineering goal of my project is to design a computational method which can efficiently and accurately predict the function of a protein given its amino acid sequence using a deep learning framework.

Proteins are sequences of amino acid residues which perform most of the important biological functions within living organisms. Even though there are several hundred known naturally occuring amino acids, only 20 of them are encoded directly by triplet codons (a sequence of 3 amino acids) in the genetic code. Different proteins perform different functions within an organism. Thus proteins are grouped into families where a family of proteins are evolutionarily related to each other and thus perform similar function.

The primary sequence of a protein the sequence of amino acids encoded in DNA. The main motivation for protein family identification methods is the fact that there are large number of known proteins but much less information is available about their function. Hence it is very desirable to have a computational method to identify the function of a protein given its primary sequence alone.

Deep learning techniques have been used successfully in recent years in the domain of Natural Language Processing (NLP). In particular, neural networks have been used effectively for various NLP tasks such as sentence classification, predictive text input and sentiment classification. It has been observed that they are many parallels between computational tasks in the domains of NLP and bioinformatics. For example, the primary sequence of a protein is similar to a sentence in a natural language and subsequences are similar to words.

Recurrent Neural Networks (RNN) (Karpathy, 2015) are particularly well suited for modeling sequential phenomenon such as sentences in a natural language or a sequence of residues in a protein. The input to the RNNs should be in the form of real valued vectors. Hence we need a mechanism to represent protein sequences of variable lengths as real valued vectors of fixed dimension. In other words, each protein primary sequence should be mapped to a sequence of vectors x_t such that $x_t \in \mathbb{R}^n$ where $\mathbb{R}$ is the set of real numbers.

Distributed Representation (Asgari & Mofrad, 2015) is one such mechanism to encode and store information about an item (such as a protein sequence) in a set of items by establishing its interactions with other items in the system. The primary goal of such as mechanism is to represent two similar items with vectors which are close to each other (meaning that the square root of the dot product of the two vectors is very close to zero).

There are two popular models to learn the distributed representation of entities (such as words or sub sequence of a protein) from a large corpus of data. One method is known as word2vec (Mikolov et al, 2013) which is a predictive model which uses a shalow 2 layer neural network to train the vector embeddings on a large corpus. The other model is known as GloVe (Global vectors for word representation) (Pennington et al, 2014) which is an unsupervised training algorithm which trains on aggregated global co-occurance statistics between every pair of words in a large text corpus.

In my project, I proposes to use a variant of the word2vec model known as protein vectors (ProtVec) described in (Asgari & Mofrad, 2015).

In summary, there are two main sub tasks in my project:

1. Train the protein vectors as described in (Asgari & Mofrad, 2015) on a large database of protein

sequences.

2. Use the protein vectors as input to a RNN to train a classifier which predicts the family of a given protein sequence.

Design Criteria: Design criteria define the product's required performance . Examples: " It will have a minimum speed of 10 KPH", The output will be within 15% of the mean of the experimental data". "It must withstand 15 repetitions of a 10N impact" The International System of units (SI) required.

My Project Design Criteria are the following:

The design criteria are as follows:

1. For both the training phases (training the distributed representation) and training the RNN, one should be able to train on large database of protein sequences (in the order of about a million sequences).
2. The testing phase consists of using the trained RNN to predict the family label for a given protein sequence. The time taken to predict the label should be in the order of a fraction of a second.
3. A popular measure in statistics to evaluate the performance of a classifier is the F1 score (which is explained in more detail below in testing and analysis section). For my project, the F1 score of the classifier should be greater than 0.95.

Constraints: Constraints are factors that limit the engineer's flexibility such as size, cost, and time limitations. Examples: "It must fit in a box no larger than 10x20x50 cm" "The maximum cost is $50" "The software must run in real time on a Raspberry Pi"

My Project Constraints are the following:

The design constraints are:

1. For the first sub task which is the training to learn the distributed representation (the protein vectors) from the protein database and the second sub task which is the training of the RNN to learn the protein classes, it is expected that the training will take a few hours of time to complete. The amount of training time depends on the type of hardware used to run the software on. But this is acceptable because both of these sub tasks are one time in nature and what is more important is the performance of the trained classifier.

Provide your chosen design. For hardware, provide a sketch. For software, provide a flowchart. Indicate the components you will develop, and the libraries you are using.

My Project Design is shown below: insert photos, diagrams, or illustrations below.

Materials:

- Python programming environment on Mac computer.
- TensorFlow machine learning package (Abadi, 2016).
- The database of annotated protein sequences from Universal Protein Resource (UniProt) database (The UniProt Consortium, 2017).

Methods:

Construction of Protein Vectors

This is the first sub task of my project. The goal of this sub task is to map each protein sequence into a set of real valued vectors of a high dimension (usually around 100). I follow the approach described in (Asgari & Mofrad, 2015).

Each protein sequence can be viewed as a sentence of a natural language. Just like a sentence is broken into a sequence of words, each protein sequence should be split into a sequence of subsequences. The most common approach is known as n-gram method. For example, for n = 3, each word consists of a sub sequence of 3 residues. Hence, each n-gram is mapped to a real valued vector. The basic idea behind the mapping is that n-grams which frequently occur together in protein sequences should be mapped to vectors which are close (as in eucledian distance) to each other. In order to observe sufficient contexts for different pairs of n-grams, we need a large database of protein sequences. I plan to use all the sequences in the UniProt database to generate the distributed representations for the n-grams.

In order to train the distributed representation of protein sequences, a neural network model known as skip gram neural network is used which is described below.

1. Each protein sequence in the database is broken into n lists of shifted but non-overlapping words (sub sequences of residues) where each sub sequence has a length of n residues. For example, for n =3, let us consider the protein sequence:MQNPLPEVMSPEHDKRTTTPMSKEANKF..

   This is split into 3 lists:

   MQN PLP EVM SPE HDK RTT TPM SKE ANF …

   QNP LPE VMS PEH DKR TTT PMS KEA NKF …

   NPL PEV MSP EHD KRT TTP MSK EAN …

   Hence for a total of N protein sequences, we generate a set of 3 * N lists of words where a word is a 3-gram like MQN, LPE etc. Each such word or 3-gram is mapped to a real valued vector as described in the next step.

2. In order to generate the vector embedding for each n-gram, a skip-gram neural network is used. A skip-gram neural network is a simple neural network with a single hidden layer. The input to this neural network is a word encoded as a one hot vector. This means that if we have a vocabulary of 1000 words, the 900^{th} word is a vector of 0's except for a 1 in the 900^{th} position. After the neural nework is trained on all the sequences in step 1, the hidden layer's weights are the real valued vectors for each of the words in the training sequences. The skip gram neural network attempts to maximize the probability of observed sequences of n-grams in the protein sequences. In other words, for a given training sequence of n-grams, the skip gram neural network computes the real valued n-dimentional vectors for its hidden layer which maximizes the following average log probability function:

$$\frac{1}{N}\sum_{t=1}^{N}\sum_{j=-c}^{c}\log Pr(w_t|w_{1..t-1})$$

where 2*c is the size of the context window for each n-gram and

$Pr(w_t|w_{1..t-1})$

is the probability of n-gram w_t following $w_{1..t-1}$.

Training the Recurrent Neural Network (RNN)

Recurrent Neural Networks (RNN) are particularly well suited for sequence prediction. I plan to make use of a particular variant of RNN known as long short term memory (LSTM) network.

The inputs to LSTM are the word vectors generated in the previous section. That means, each protein sequence to be classified is converted into a sequence of overlapping n-grams and the vector embedding for each n-gram is used as an input to the LSTM. The output of the LSTM is a probability distribution over the next word in the sequence.

A RNN is a type of neural network particularly well-suited for modeling sequential phenomenon such as the sequence of words in a sentence of a natural language. At each time step t , an RNN takes the input vector $x_t \in \mathbb{R}^n$ and the hidden state vector $h_{t-1} \in \mathbb{R}^m$ and computes the next hidden state h_t by computing the following function:

$$h_t = f(Wx_t + Uh_{t-1} + b)$$

where W , U and b are the parameters of the RNN to be learned from training and f is an element-wise non-linearity.

$W \in \mathbb{R}^{m \times n}$ Is the input-to-hidden transformation.

$U \in \mathbb{R}^{m \times n}$ Is the hidden-to-hidden transformation.

$b \in \mathbb{R}^m$ Is the bias term.

In the context of protein class prediction, the input vector x_t is the word embedding of the input word (n-gram) at time t which is a real valued vector in $\mathbb{R}^n$.

RNNs have a problem of learning long range dependencies. I propose to use a variant of RNN known as Long Short-Term Memory (LSTM) network which addesses this problem by adding a memory cell vector $c_t \in \mathbb{R}^n$ at each time step t .

Let $[w_1, w_2, \ldots, w_t]$ be the sequence of n-grams input so far till time t . Then the following formula estimates the probability of the next n-gram w_{t+1} being the n-gram with index j :

$$Pr(w_{t+1}=j|w_{1..t})=\frac{\exp(h_t.\,p^j+q^j)}{\sum_{j'\in V}\exp(h_t.\,p^{j'}+q^{j'})}$$

where V is the vocabulary of n-grams of fixed size. p^j Is the jth column of the output embedding matrix $P\in\mathbb{R}^{m\times|(V)|}$ and q^j is a bias term.

For training protein sentence $[w_1, w_2, ..., w_T]$, training the LSTM involves minimizing the negative log-likelihood of the training sequence as below.

$$-\sum_{t=1}^{T}\log Pr(w_t|w_{1..t-1})$$

The output of the LSTM is a probability distribution over the next possible n-gram. We use a softmax layer to convert that into a protein class label.

Test and evaluate your prototypes against the design criteria listed above to show how well the product meets the need/goal. Provide a test plan describing how you will test the design criteria and constraints you listed above., How will you analyze the data? If the product requires human testing please fill out and append https://science-fair.org/wp/wp-content/uploads/2015/10/Research-Plan-Human-Participants.docx

I test and analyze my prototypes using the following methods:

For the first sub task of my project which is the task of generating the protein vectors I plan to use all the sequences in the UniProt database and generate the vector embeddings for each n-gram.

For the second sub task, the protein sequences in the UniProt database are divided into 3 parts:

1. The training set. This is the set of sequences used to train the RNN.
2. The hold-out set. This is the set of sequeneced that is used to fine tune the configuration of RNN.
3. The test set. This is the set used to evaluate the performance of the classification of the RNN. I plan to use the F1 score which is defined as the harmonic mean of the precision and recall:

$$F1=\frac{2*(precision.recall)}{(precision+recall)}$$ where

$$precision=\frac{truePositives}{(truePositives+falsePositives)}$$ and

$$recall=\frac{truePositives}{(truePositives+falseNegatives)}$$

Code to generate Word Vectors

#

#

#
==========
=

"""Basic word2vec example."""

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import collections
import math
import os
import sys
import argparse
```

```
import random
from tempfile import gettempdir
import zipfile

import numpy as np
from six.moves import urllib
from six.moves import xrange  # pylint: disable=redefined-builtin
import tensorflow as tf

from tensorflow.contrib.tensorboard.plugins import projector

# Give a folder path as an argument with '--log_dir' to save
# TensorBoard summaries. Default is a log folder in current directory.
current_path = os.path.dirname(os.path.realpath(sys.argv[0]))

parser = argparse.ArgumentParser()
parser.add_argument(
    '--log_dir',
    type=str,
    default=os.path.join(current_path, 'log1'),
    help='The log directory for TensorBoard summaries.')
FLAGS, unparsed = parser.parse_known_args()

# Create the directory for TensorBoard variables if there is not.
if not os.path.exists(FLAGS.log_dir):
  os.makedirs(FLAGS.log_dir)

# Step 1: Download the data.
```

```
filename = "uniprot_sprot.fasta"



# Read the data into a list of strings.
def read_data(filename):
  """Extract the first file enclosed in a zip file as a list of words."""
  with zipfile.ZipFile(filename) as f:
    data = tf.compat.as_str(f.read(f.namelist()[0])).split()
  return data


def get_protein_sequences(file_name):
    protein_sequences = list()
    prot = ""
    with open(file_name, "r") as fp:
        for line in fp:
            if line.startswith(">sp"):
                if prot != "":
                    protein_sequences.append(prot)
                    prot = ""
            else:
                prot += line.strip()


    if prot != "":
        protein_sequences.append(prot)
    return protein_sequences
```

```
protein_sequences = get_protein_sequences(filename)
print('Number of sequences:', len(protein_sequences))

# Step 2: Build the dictionary and replace rare words with UNK token.



def build_dataset(sequences):
    word_dictionary = dict()
    word_index = 0
    data = list()

    for seq in sequences:
        seq_len = len(seq)
        num_words = int(seq_len/3)
        for word_idx in range(0, num_words):
            word = seq[word_idx * 3: word_idx * 3 + 3]
            if word not in word_dictionary:
                word_dictionary[word] = word_index
                word_index = word_index + 1
                index = word_index
            else:
                index = word_dictionary.get(word, 0)
            data.append(index)

    reversed_dictionary = dict(zip(word_dictionary.values(), word_dictionary.keys()))
    return data, word_dictionary, reversed_dictionary
```

```
# Filling 4 global variables:
# data - list of codes (integers from 0 to vocabulary_size-1).
#   This is the original text but words are replaced by their codes
# count - map of words(strings) to count of occurrences
# dictionary - map of words(strings) to their codes(integers)
# reverse_dictionary - maps codes(integers) to words(strings)
data, dictionary, reverse_dictionary = build_dataset(
    protein_sequences)
#del vocabulary  # Hint to reduce memory.
#print('Most common words (+UNK)', count[:5])
print('Sample data', data[:10], [reverse_dictionary[i] for i in data[:10]])
vocabulary_size = len(dictionary)
print(vocabulary_size)
data_index = 0


# Step 3: Function to generate a training batch for the skip-gram model.
def generate_batch(batch_size, num_skips, skip_window):
  global data_index
  assert batch_size % num_skips == 0
  assert num_skips <= 2 * skip_window
  batch = np.ndarray(shape=(batch_size), dtype=np.int32)
  labels = np.ndarray(shape=(batch_size, 1), dtype=np.int32)
  span = 2 * skip_window + 1  # [ skip_window target skip_window ]
  buffer = collections.deque(maxlen=span)
  if data_index + span > len(data):
    data_index = 0
```

```
  buffer.extend(data[data_index:data_index + span])
  data_index += span
  for i in range(batch_size // num_skips):
    context_words = [w for w in range(span) if w != skip_window]
    words_to_use = random.sample(context_words, num_skips)
    for j, context_word in enumerate(words_to_use):
      batch[i * num_skips + j] = buffer[skip_window]
      labels[i * num_skips + j, 0] = buffer[context_word]
    if data_index == len(data):
      buffer.extend(data[0:span])
      data_index = span
    else:
      buffer.append(data[data_index])
      data_index += 1
  # Backtrack a little bit to avoid skipping words in the end of a batch
  data_index = (data_index + len(data) - span) % len(data)
  return batch, labels


batch, labels = generate_batch(batch_size=8, num_skips=2, skip_window=1)
for i in range(8):
  print(batch[i], reverse_dictionary[batch[i]], '->', labels[i, 0],
        reverse_dictionary[labels[i, 0]])

# Step 4: Build and train a skip-gram model.

batch_size = 128
embedding_size = 128  # Dimension of the embedding vector.
```

```
skip_window = 2  # How many words to consider left and right.
num_skips = 2  # How many times to reuse an input to generate a label.
num_sampled = 64  # Number of negative examples to sample.

# We pick a random validation set to sample nearest neighbors. Here we limit the
# validation samples to the words that have a low numeric ID, which by
# construction are also the most frequent. These 3 variables are used only for
# displaying model accuracy, they don't affect calculation.
valid_size = 16  # Random set of words to evaluate similarity on.
valid_window = 100  # Only pick dev samples in the head of the distribution.
valid_examples = np.random.choice(valid_window, valid_size, replace=False)

graph = tf.Graph()

with graph.as_default():

  # Input data.
  with tf.name_scope('inputs'):
    train_inputs = tf.placeholder(tf.int32, shape=[batch_size])
    train_labels = tf.placeholder(tf.int32, shape=[batch_size, 1])
    valid_dataset = tf.constant(valid_examples, dtype=tf.int32)

  # Ops and variables pinned to the CPU because of missing GPU implementation
  with tf.device('/cpu:0'):
    # Look up embeddings for inputs.
    with tf.name_scope('embeddings'):
      embeddings = tf.Variable(
          tf.random_uniform([vocabulary_size, embedding_size], -1.0, 1.0))
```

```
    embed = tf.nn.embedding_lookup(embeddings, train_inputs)

  # Construct the variables for the NCE loss
  with tf.name_scope('weights'):
    nce_weights = tf.Variable(
        tf.truncated_normal(
            [vocabulary_size, embedding_size],
            stddev=1.0 / math.sqrt(embedding_size)))
  with tf.name_scope('biases'):
    nce_biases = tf.Variable(tf.zeros([vocabulary_size]))

# Compute the average NCE loss for the batch.
# tf.nce_loss automatically draws a new sample of the negative labels each
# time we evaluate the loss.
# Explanation of the meaning of NCE loss:
#   http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/
with tf.name_scope('loss'):
  loss = tf.reduce_mean(
      tf.nn.nce_loss(
          weights=nce_weights,
          biases=nce_biases,
          labels=train_labels,
          inputs=embed,
          num_sampled=num_sampled,
          num_classes=vocabulary_size))

# Add the loss value as a scalar to summary.
tf.summary.scalar('loss', loss)
```

```
  # Construct the SGD optimizer using a learning rate of 1.0.
  with tf.name_scope('optimizer'):
    optimizer = tf.train.GradientDescentOptimizer(1.0).minimize(loss)

  # Compute the cosine similarity between minibatch examples and all embeddings.
  norm = tf.sqrt(tf.reduce_sum(tf.square(embeddings), 1, keep_dims=True))
  normalized_embeddings = embeddings / norm
  valid_embeddings = tf.nn.embedding_lookup(normalized_embeddings,
                                            valid_dataset)
  similarity = tf.matmul(
      valid_embeddings, normalized_embeddings, transpose_b=True)

  # Merge all summaries.
  merged = tf.summary.merge_all()

  # Add variable initializer.
  init = tf.global_variables_initializer()

  # Create a saver.
  saver = tf.train.Saver()

# Step 5: Begin training.
num_steps = 100001

with tf.Session(graph=graph) as session:
  # Open a writer to write summaries.
  writer = tf.summary.FileWriter(FLAGS.log_dir, session.graph)
```

```
# We must initialize all variables before we use them.
init.run()
print('Initialized')

average_loss = 0
for step in xrange(num_steps):
  batch_inputs, batch_labels = generate_batch(batch_size, num_skips,
                                              skip_window)
  feed_dict = {train_inputs: batch_inputs, train_labels: batch_labels}

  # Define metadata variable.
  run_metadata = tf.RunMetadata()

  # We perform one update step by evaluating the optimizer op (including it
  # in the list of returned values for session.run()
  # Also, evaluate the merged op to get all summaries from the returned "summary" variable.
  # Feed metadata variable to session for visualizing the graph in TensorBoard.
  _, summary, loss_val = session.run(
      [optimizer, merged, loss],
      feed_dict=feed_dict,
      run_metadata=run_metadata)
  average_loss += loss_val

  # Add returned summaries to writer in each step.
  writer.add_summary(summary, step)
  # Add metadata to visualize the graph for the last run.
  if step == (num_steps - 1):
```

```
    writer.add_run_metadata(run_metadata, 'step%d' % step)

  if step % 2000 == 0:
    if step > 0:
      average_loss /= 2000
    # The average loss is an estimate of the loss over the last 2000 batches.
    print('Average loss at step ', step, ': ', average_loss)
    average_loss = 0

  # Note that this is expensive (~20% slowdown if computed every 500 steps)
  if step % 10000 == 0:
    sim = similarity.eval()
    for i in xrange(valid_size):
      valid_word = reverse_dictionary[valid_examples[i]]
      top_k = 8  # number of nearest neighbors
      nearest = (-sim[i, :]).argsort()[1:top_k + 1]
      log_str = 'Nearest to %s:' % valid_word
      for k in xrange(top_k):
        close_word = reverse_dictionary[nearest[k]]
        log_str = '%s %s,' % (log_str, close_word)
      print(log_str)
final_embeddings = normalized_embeddings.eval()

# Write corresponding labels for the embeddings.
with open(FLAGS.log_dir + '/metadata.tsv', 'w') as f:
  for i in xrange(vocabulary_size):
    f.write(reverse_dictionary[i] + '\n')
```

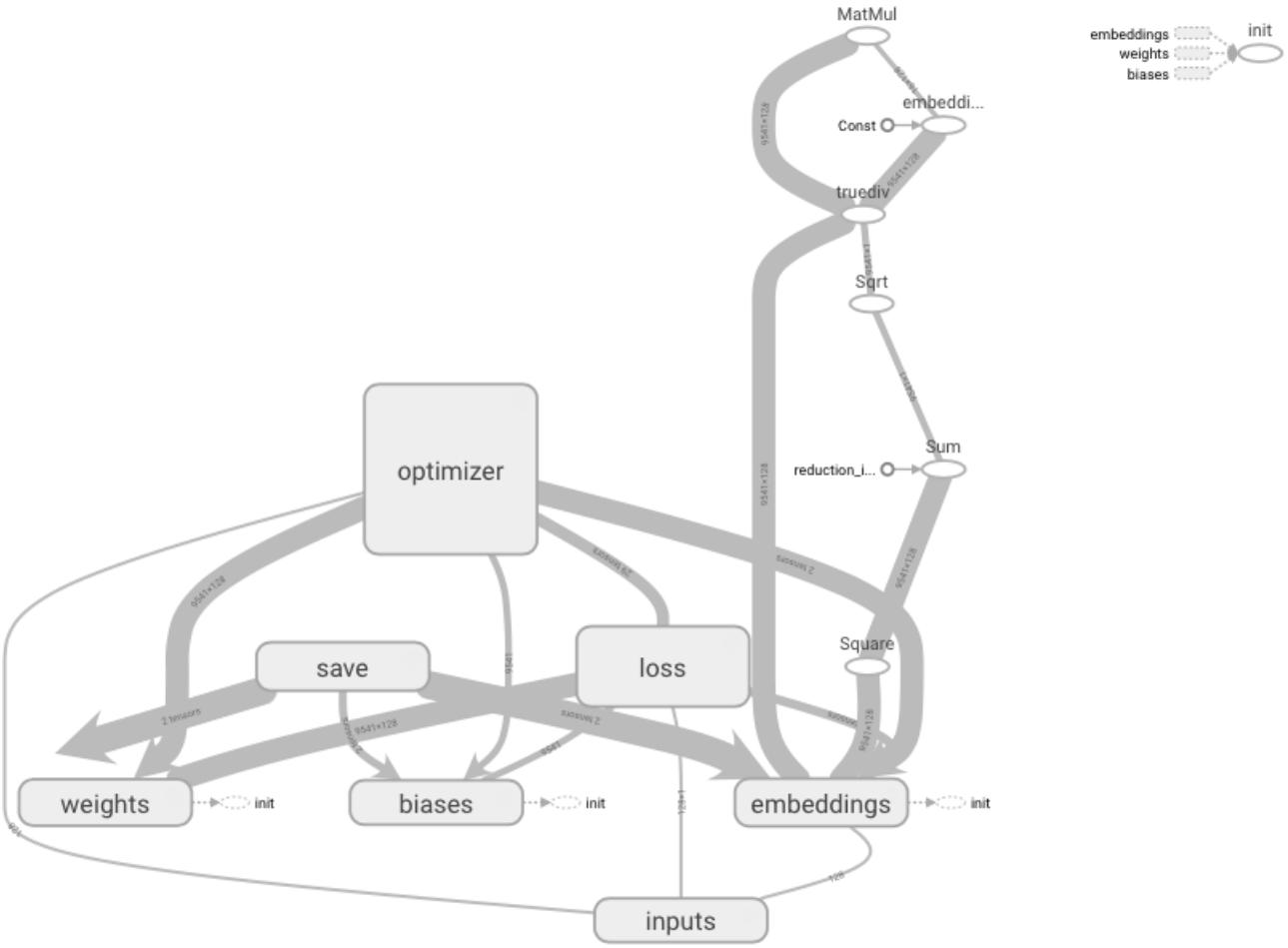
```
  # Save the model for checkpoints.
  saver.save(session, os.path.join(FLAGS.log_dir, 'model.ckpt'))

  # Create a configuration for visualizing embeddings with the labels in TensorBoard.
  config = projector.ProjectorConfig()
  embedding_conf = config.embeddings.add()
  embedding_conf.tensor_name = embeddings.name
  embedding_conf.metadata_path = os.path.join(FLAGS.log_dir, 'metadata.tsv')
  projector.visualize_embeddings(writer, config)

writer.close()

# Step 6: Visualize the embeddings.


# pylint: disable=missing-docstring
# Function to draw visualization of distance between embeddings.
def plot_with_labels(low_dim_embs, labels, filename):
  assert low_dim_embs.shape[0] >= len(labels), 'More labels than embeddings'
  plt.figure(figsize=(18, 18))  # in inches
  for i, label in enumerate(labels):
    x, y = low_dim_embs[i, :]
    plt.scatter(x, y)
    plt.annotate(
        label,
        xy=(x, y),
        xytext=(5, 2),
        textcoords='offset points',
```

```
        ha='right',
        va='bottom')

  plt.savefig(filename)



try:
  # pylint: disable=g-import-not-at-top
  from sklearn.manifold import TSNE
  import matplotlib.pyplot as plt

  tsne = TSNE(
      perplexity=30, n_components=2, init='pca', n_iter=5000, method='exact')
  plot_only = 500
  low_dim_embs = tsne.fit_transform(final_embeddings[:plot_only, :])
  labels = [reverse_dictionary[i] for i in xrange(plot_only)]
  plot_with_labels(low_dim_embs, labels, os.path.join(gettempdir(), 'tsne.png'))

except ImportError as ex:
  print('Please install sklearn, matplotlib, and scipy to show embeddings.')
  print(ex)
  Code to generate test and train data
import collections
import re

def get_protein_families(file_name):
    family_dict = dict()
    id_dict = dict()
```

```
family_id = 0
with open(file_name, "r") as fp:
    cnt = 0
    id_list = list()
    for line in fp:
        if 'family' in line:
            if cnt > 200:
                id_dict[family_id] = id_list
                #print(len(id_list))
            family_id = family_id + 1
            cnt = 0
            #id_list[:] = []
            id_list = list()
        else:
            match = re.search(r'.+\((.+)\).+\((.+)\).+\((.+)\).*', line)
            if match:
                family_dict[match.group(1)] = family_id
                family_dict[match.group(2)] = family_id
                family_dict[match.group(3)] = family_id
                cnt = cnt + 3
                id_list.append(match.group(1))
                id_list.append(match.group(2))
                id_list.append(match.group(3))
            else:
                match = re.search(r'.+\((.+)\).+\((.+)\).*', line)
                if match:
                    family_dict[match.group(1)] = family_id
                    family_dict[match.group(2)] = family_id
```

```
                cnt = cnt + 2
                id_list.append(match.group(1))
                id_list.append(match.group(2))
            else:
                match = re.search(r'.+\((.+)\).*', line)
                if match:
                    family_dict[match.group(1)] = family_id
                    cnt = cnt + 1
                    id_list.append(match.group(1))



    return id_dict


def get_protein_sequences(file_name):
    seq_dict = dict()
    prot = ""
    with open(file_name, "r") as fp:
        for line in fp:
            if line.startswith(">sp"):
                match = re.search(r'>sp\|(.+)\|.+', line)
                if prot != "":
                    seq_dict[seq_id] = prot
                    prot = ""
                seq_id = match.group(1)
            else:
                prot += line.strip()
    if prot != "":
        seq_dict[seq_id] = prot
```

```
    return seq_dict


seq_dict = get_protein_sequences('uniprot_sprot.fasta')
id_dict = get_protein_families('similar.txt')


with open('train.txt', "w") as train_fp:
    with open('test.txt', "w") as test_fp:
        for family_id, id_list in id_dict.items():
            num_seq = len(id_list)
            print(num_seq)
            train_seq_num = num_seq * 0.6
            idx = 0
            for seq_id in id_list:
                if idx < train_seq_num:
                    train_fp.write(seq_dict[seq_id])
                    train_fp.write(" ")
                    train_fp.write(str(family_id))
                    train_fp.write("\n")
                else:
                    test_fp.write(seq_dict[seq_id])
                    test_fp.write(" ")
                    test_fp.write(str(family_id))
                    test_fp.write("\n")
                idx = idx + 1
```


MatMul
embeddi...
Const
truediv
Sqrt
Sum
reduction_i...
Square
optimizer
save
loss
weights
init
biases
init
embeddings
init
inputs
embeddings
weights
biases
init

Number of Amino Acid Sequences	556825
Number of Tri-Grams of Amino Acids	9541
Number of Dimensions of Embedding Vector	128
Training Batch Size	128
Skip Window Size	5 (2 left 2 right)
Number of Skips	2

```
import tensorflow as tf
from tensorflow.contrib import rnn
import numpy as np

dictionary = np.load('dictionary.npy').tolist()
embeddings = np.load('embeddings.npy')
numDimensions = len(embeddings[0])
batchSize = 2
lstmUnits = 64
numClasses = 2
iterations = 10
maxSeqLength = 10

tf.reset_default_graph()
```

```
labels = tf.placeholder(tf.float32, [batchSize, numClasses])

input_data = tf.placeholder(tf.int32, [batchSize, maxSeqLength])


data = tf.Variable(tf.zeros([batchSize, maxSeqLength, numDimensions]),dtype=tf.float32)

data = tf.nn.embedding_lookup(embeddings ,input_data)


lstmCell = rnn.BasicLSTMCell(lstmUnits)

lstmCell = tf.contrib.rnn.DropoutWrapper(cell=lstmCell, output_keep_prob=0.75)

value, _ = tf.nn.dynamic_rnn(lstmCell, data, dtype=tf.float32)

weight = tf.Variable(tf.truncated_normal([lstmUnits, numClasses]))

bias = tf.Variable(tf.constant(0.1, shape=[numClasses]))

value = tf.transpose(value, [1, 0, 2])

last = tf.gather(value, int(value.get_shape()[0]) - 1)

prediction = (tf.matmul(last, weight) + bias)

correctPred = tf.equal(tf.argmax(prediction,1), tf.argmax(labels,1))

accuracy = tf.reduce_mean(tf.cast(correctPred, tf.float32))

loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
labels=labels))

optimizer = tf.train.AdamOptimizer().minimize(loss)


sess = tf.InteractiveSession()

saver = tf.train.Saver()

sess.run(tf.global_variables_initializer())

def getTrainBatch():

   batch = [[2, 45, 654, 33, 345, 6665, 3, 5, 4, 654],[21, 425, 6524, 303, 3425, 16665, 4443, 35, 14,
11]]

   labels=[[0,1],[1,0]]

   return batch, labels
```

```
for i in range(iterations):
    nextBatch, nextBatchLabels = getTrainBatch()
    sess.run(optimizer, {input_data: nextBatch, labels: nextBatchLabels})
```

Materials

Type	Name	Description	Location
Software	Python	Python Programming Environment on Mac Computer	www.python.org
Software	TensorFlow	Open Source Machine Learning Framework	www.tensorflow.org
Software	NumPy	Python package for Scientific Computing	www.numpy.org
Data	UniProt : Swiss-Prot	A reviewed, annotated database of protein sequences. Each protein sequence is identified by a unique key and is followed by the actual protein sequence of amino acids. File name:uniprot_sprot.fasta	https://www.uniprot.org/uniprot/query=reviewed:yes
Data	UniProt - Swiss-Prot Protein Knowledgebase	A database of protein families. The file lists for each family a list of sequence ids of protein sequences belonging to that family. File Name:similar.txt	https://www.uniprot.org/docs/similar

Bibliography

1. Abadi, M. (2016). TensorFlow: Learning functions at scale. *ACM SIGPLAN Notices*, *51*(9), 1–1. doi:10.1145/3022670.2976746

2. Asgari, E., & Mofrad, M. R. (2015). Continuous Distributed Representation of Biological Sequences for Deep Proteomics and Genomics. *Plos One,10*(11). doi:10.1371/journal.pone.0141287

3. Bengio, Y., Ducharme, R., Vincent, P., & Jaivin, C. (2003). A Neural Probabilistic Language Model. *Journal of Machine Learning Research, 3*.

4. Britz, Denny. "Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs." WildML. July 08, 2016. Accessed January 26, 2017. http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/.

5. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. Cambridge, MA: The MIT Press.

6. Karpathy, A. (n.d.). The Unreasonable Effectiveness of Recurrent Neural Networks. Retrieved January 24, 2018, from http://karpathy.github.io/2015/05/21/rnn-effectiveness/

7. Lee, T. K., & Nguyen, T. (2016, June 19). Protein Family Classification with Neural Networks. Retrieved January 26, 2018, from https://cs224d.stanford.edu/reports/LeeNguyen.pdf

8. Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

9. Olah, C. (n.d.). Understanding LSTM Networks. Retrieved January 26, 2017, from http://colah.github.io/posts/2015-08-Understanding-LSTMs/

10. Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global Vectors for Word Representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. doi:10.3115/v1/d14-1162

11. Recurrent Neural Networks | TensorFlow. (n.d.). Retrieved January 26, 2017, from **https://www.tensorflow.org/tutorials/recurrent/**

12. Ruder, S. (2016, November 22). On word embeddings - Part 1. Retrieved January 26, 2017, from http://sebastianruder.com/word-embeddings-1/

13. The UniProt Consortium. UniProt: the universal protein knowledgebase . *Nucleic Acids Research*, Volume 45, Issue D1, 4 January 2017, Pages D158–D169