Chapter_7_Section_1_Autoencoder

February 3, 2019

0.1 Ch 07: Concept 01

0.2 Autoencoder

All we'll need is TensorFlow and NumPy:

```
In [1]: import tensorflow as tf
        import numpy as np
```

Define the autoencoder class:

```
In [2]: import tensorflow as tf
        import numpy as np
        import datetime as dt

        def get_batch(X, size):
            """Instead of feeding all the training data to the training op, we will feed data
            a = np.random.choice(len(X), size, replace=False)
            return X[a]

        class Autoencoder:
            def __init__(self, input_dim, hidden_dim, epoch=1000, batch_size=50, learning_rate=
                # In the construct we can define eveything that doesn't need a tf session to b
                self.epoch = epoch
                self.batch_size = batch_size
                self.learning_rate = learning_rate
                # Input placeholder
                x = tf.placeholder(dtype=tf.float32, shape=[None, input_dim])
                # *** TENSORBOARD ***
                # define the encode/decode variables under their own scopes
                # for better visualisation in Tensorboard
                # *** TENSORBOARD ***
                with tf.name_scope('encode'):
                    weights = tf.Variable(tf.random_normal([input_dim, hidden_dim], dtype=tf.fl
                    biases = tf.Variable(tf.zeros([hidden_dim]), name='biases')
                    encoded = tf.nn.sigmoid(tf.matmul(x, weights) + biases)
                with tf.name_scope('decode'):
                    weights = tf.Variable(tf.random_normal([hidden_dim, input_dim], dtype=tf.fl
```

```
            biases = tf.Variable(tf.zeros([input_dim]), name='biases')
            decoded = tf.matmul(encoded, weights) + biases
        # set as class properies/methods
        self.x = x
        self.encoded = encoded
        self.decoded = decoded
        # loss
        self.loss = tf.sqrt(tf.reduce_mean(tf.square(tf.subtract(self.x, self.decoded))
        # *** TENSORBOARD ***
        # add a summary tensor to collect the loss
        self.loss_summ = tf.summary.scalar('loss', self.loss)
        # *** TENSORBOARD ***
        # optimiser
        self.train_op = tf.train.RMSPropOptimizer(self.learning_rate).minimize(self.los
        # model saver
        self.saver = tf.train.Saver()

    def train(self, data):
        with tf.Session() as sess:
            sess.run(tf.global_variables_initializer())
            sess.run(tf.local_variables_initializer())
            # *** TENSORBOARD ***
            # set directory to collect saved summary tensors with each run
            # based on run time
            now = dt.datetime.now()
            currentDir = "./logs/" + now.strftime("%Y%m%d-%H%M%S") + "/"
            # create writer and set directory and graph
            writer = tf.summary.FileWriter(currentDir, graph=sess.graph)
            # save graph in the PARENT directory of logs
            # this looks like it's not needed...
            # tf.train.write_graph(sess.graph_def, currentDir, 'graph.pbtxt')
            # *** TENSORBOARD ***
            # iterate over every epoch
            for i in range(self.epoch):
                # iterate over every batch
                for j in range(np.shape(data)[0] // self.batch_size):
                    batch_data = get_batch(data, self.batch_size)
                    l, _, l_summ = sess.run([self.loss, self.train_op, self.loss_summ]
                                            feed_dict={self.x: batch_data})
                    # *** TENSORBOARD ***
                    # record loss with each batch
                    # writer.add_summary(summary=l_summ, global_step=i)
                    # writer.flush()
                    # *** TENSORBOARD ***
                if i % 100 == 0:
                    print('epoch {0}: loss = {1}'.format(i, l))
                    self.saver.save(sess, './model.ckpt')
                    # *** TENSORBOARD ***
```

```
                        # every 10 epochs
                        writer.add_summary(summary=l_summ, global_step=i)
                        writer.flush()
                        # *** TENSORBOARD ***
                    # *** TENSORBOARD ***
                    # record loss with each epoch
                    # writer.add_summary(summary=l_summ, global_step=i)
                    # writer.flush()
                    # *** TENSORBOARD ***
                # save model
                self.saver.save(sess, './model.ckpt')
                # close writer
                writer.close()

    def test(self, data):
        # load model
        with tf.Session() as sess:
            self.saver.restore(sess, './model.ckpt')
            # run test data through encoder and decoder
            hidden, reconstructed = sess.run([self.encoded, self.decoded], feed_dict={s
        print('input', data)
        print('compressed', hidden)
        print('reconstructed', reconstructed)
        return reconstructed
```

The *Iris dataset* is often used as a simple training dataset to check whether a classification algorithm is working. The sklearn library comes with it, `pip install sklearn`.

```
In [3]: from sklearn import datasets

        # hidden dimensions
        hidden_dim = 1
        # load only the feature data from the Iris data set
        data = datasets.load_iris().data
        # the dimensions of the input data, for Iris it's 4
        input_dim = len(data[0])
        # create an instance of the autoencoder with the necessary dimensions
        ae = Autoencoder(input_dim, hidden_dim)
        ae.train(data)
        ae.test([[8, 4, 6, 2]])
```

```
epoch 0: loss = 3.8478190898895264
epoch 100: loss = 3.721243381500244
epoch 200: loss = 3.223569393157959
epoch 300: loss = 2.8416402339935303
epoch 400: loss = 2.2708091735839844
epoch 500: loss = 1.8190032243728638
epoch 600: loss = 1.4302263259887695
```

```
epoch 700: loss = 1.1656414270401
epoch 800: loss = 1.0143120288848877
epoch 900: loss = 0.9401819109916687
INFO:tensorflow:Restoring parameters from ./model.ckpt
input [[8, 4, 6, 2]]
compressed [[0.97981286]]
reconstructed [[6.7325454 3.0909522 4.3031726 1.5267226]]


Out[3]: array([[6.7325454, 3.0909522, 4.3031726, 1.5267226]], dtype=float32)
```