TensorFlow

Ardelean Eugen-Richard

May 23, 2018

Contents

1 Overview

1.1 Running instructions

I have chosen to install TensorFlow with CPU support on Windows with Anaconda[1]. For this, I have followed these steps:

1. Download and install Anaconda, from their site.
2. Create a conda environment named tensorflow by invoking the following command:
   C:>conda create -n tensorflow pip python=3.5
3. Activate the conda environment by issuing the following command:
   C:>activate tensorflow
4. Install TensorFlow inside your conda environment, by entering the following command:
   (tensorflow)C:>pip install –ignore-installed –upgrade tensorflow

To verify that the installation is successful, follow these steps:

1. Open the anaconda environment and invoke python by typing the following command:
   python

2. Enter this short program
   >>>import tensorflow as tf
   >>>hello = tf.constant('Hello, TensorFlow!')
   >>>sess = tf.Session()
   >>>print(sess.run(hello))

3. Verify if the output looks like this: Hello, TensorFlow!

1.2 Theoretical aspects

1.2.1 Data representation

TensorFlow is an open-source software library for dataflow programming across a range of tasks. It is a symbolic math library, and also used for machine learning applications such as neural networks.

The name TensorFlow is derived from the operations which neural networks perform on multidimensional data arrays or tensors! Its literally a flow of tensors.

In tensorflow, data is represented in the form of tensors, which are multidimensional data arrays, they can take the form of vectors, matrices and n-dimension arrays.

Rank defines the dimension:
Rank 0: scalar
Rank 1: vector
Rand 2: matrix
...
Rank n: n-Tensor

Beside the tensors, you have the dataflow graph(computation graph), first you build this graph and then you run it.

Example:[2]:
Building the computation graph:

```
import tensorflow as tf
node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0)
```

Running a computational graph:

```
sess = tf.Session()
print(sess.run([node1, node2]))
sess.close()
```

1.2.2 Algorithm

Artificial Neural Network algorithms are inspired by the human brain. The artificial neurons are interconnected and communicate with each other. Each connection is weighted by previous learning events and with each new input of data more learning takes place. A lot of different algorithms are associated with Artificial Neural Networks and one of the most important is Deep learning.

Neural networks are one of the learning algorithms used within machine learning. They consist of different layers for analyzing and learning data.

Neural networks are based on layers, it has an input layer, an output layers, and a programmer-defined number of hidden-layers. Every hidden layer tries to detect patterns on the picture. When a pattern is detected the next hidden layer is activated and so on.

Let's say we have the picture of a car. The first hidden layer detects edges. Then the following layers combine other edges found in the data, ultimately a specified layer attempts to detect a wheel pattern or a window pattern. Depending on the amount of layers, it will be or not be able to define what is on the picture, in this case a car.The more layers in a neural network, the more is learned and the more accurate the pattern detection is. Neural Networks learn and attribute weights to the connections between the different neurons each time the network processes data. This means the next time it comes across such a picture, it will have learned that this particular section of the picture is probably associated with for example a tire or a door.

Weights are the most important factor in converting an input to impact the output. This is similar to slope in linear regression, where a weight is multiplied to the input to add up to form the output. Weights are numerical parameters which determine how strongly each of the neurons affects the other.

For a typical neuron, if the inputs are x1, x2, and x3, then the synaptic weights to be applied to them are denoted as w1, w2, and w3.

Output is

$$y = f(x) = \sum x_i * w_i$$

where i is 1 to the number of inputs.
Simply, this is a matrix multiplication to arrive at the weighted sum.

Bias is like the intercept added in a linear equation. It is an additional parameter which is used to adjust the output along with the weighted sum of the inputs to the neuron.

The processing done by a neuron is thus denoted as:
output=sum(weights*inputs)+bias

A function is applied on this output and is called an activation function. The input of the next layer is the output of the neurons in the previous layer.
If you have input X1, X2, X3, with weights W1,W2,W3 linked to the first neu-

ron of the first hidden layer (second layer overall), this layer will have the value of

$$activationFunction(\sum_{i=1}^{n=3} W_i * X_i)$$

1.3 Existing Example

Example of Linear Regression:

```
import tensorflow as tf
#model parameters
W = tf.Variable([.3], tf.float32)
b = tf.Variable([-.3], tf.float32)

#inputs and outputs
x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

linearModel = W*x+b


#loss
squaredDelta = tf.square(linearModel-y)
loss = tf.reduce_sum(squaredDelta)

#optimizer to get the loss minimized,
# parameter(learning rate)
optimizer = tf.train.GradientDescentOptimizer(0.01)
train = optimizer.minimize(loss)

init = tf.global_variables_initializer()

sess = tf.Session()
sess.run(init)
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
for i in range(1000):
    sess.run(train, {x:[1,2,3,4], y:[0,-1,-2,-3]})
print(sess.run([W,b]))
print(sess.run(loss, {x:[1,2,3,4], y:[0,-1,-2,-3]}))
sess.close()
```

For calculating the loss, we calculate the distance from our line (defined by Wx+b) to the actual values (y) which means linearModel-y, we square and add them all together, that is the loss. We observe that for values -.3 and 3 for W and b we have a loss of 23.66, while for values -1 and 1 we have a loss of 0, we want to get as close to 0 as possible.

For getting the values of W and b close to -1,1 we use an optimizer that will, based on the training data, adjust these values (gradient descent), as we can

see from the last 2 prints, the values have gotten close to -1,1 and the loss has been reduced significantly.

1.4 Your own small Example

Linear Regression on a bigger dataset, using normal equation:
It starts by fetching the dataset; then it adds an extra bias input feature (x0 = 1) to all training instances (it does so using NumPy so it runs immediately); then it creates two TensorFlow constant nodes, X and y, to hold this data and the targets, and it uses some of the matrix operations provided by TensorFlow to define theta. These matrix functions transpose(), matmul(), and matrix inverse() are self-explanatory, but as usual they do not perform any computations immediately; instead, they create nodes in the graph that will perform them when the graph is run. You may recognize that the definition of theta corresponds to the Normal Equation (

$$theta = (X^T * X)^{-1} * X^T * y$$

). Finally, the code creates a session and uses it to evaluate theta (represent model parameters).

```
import tensorflow as tf
import numpy as np
from sklearn.datasets import fetch_california_housing
housing = fetch_california_housing()
m, n = housing.data.shape
housing_data_plus_bias =
    np.c_[np.ones((m, 1)), housing.data]

X = tf.constant(housing_data_plus_bias,
                dtype=tf.float32, name="X")
y = tf.constant(housing.target.reshape(-1, 1),
                dtype=tf.float32, name="y")
XT = tf.transpose(X)

theta = tf.matmul(tf.matmul
                    (tf.matrix_inverse(
                        tf.matmul(XT, X)),
                                XT),
                    y)
with tf.Session() as sess:
    theta_value = theta.eval()
```

A more complicated example:

The data set is the MNIST Database of Handwritten Digits. This data set is very famous and it is usually considered one of the first steps to learn about computer vision.

The data consists of individual black and white handwritten digits, with the size

of 28x28 pixels. There are 55,000 digits for training, 5,000 for cross-validation and 10,000 digits for testing. All digits have a label with the true value. So this is a "Supervised machine learning problem".

```
# Data Manipulation and Visualization
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

# TensorFlow
import tensorflow as tf

# Get the MINST data. It is available from the tensorflow
# package
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/",
                                  one_hot=True)
```

Each individual digit is an array with 784 (28x28) values. Each value represents the color for one pixel.
The label format is the one-hot encoding style. This means that the label corresponds to the index of the array where the value is 1.

```
Example:
[1,2,3,4,5,6,7,8,9]
[0,0,0,0,0,0,0,1,0]
```

Input: mnist.train.labels[5]
Output: array([0., 0., 0., 0., 0., 0., 0., 1., 0.])

Neural Network:

```
# Network Parameters (784-300-10)
n_input = 784
hidden_layer_neurons = 300
n_classes = 10

# Training Parameters
learning_rate = 0.005
training_epochs = 30000
batch_size = 50
```

Creating TensorFlow Variables and Model:

```
# x and y placeholders
x = tf.placeholder("float", [None, n_input])
y = tf.placeholder("float", [None, n_classes])

# Create weights and biases
```

```
# that will be used in the neural network
w1 = tf.Variable(tf.random_normal([n_input,
                                   hidden_layer_neurons]))
w2 = tf.Variable(tf.random_normal([hidden_layer_neurons,
                                      n_classes]))
b1 = tf.Variable(tf.random_normal([hidden_layer_neurons]))
b2 = tf.Variable(tf.random_normal([n_classes]))

# The multilayer perceptron model
hidden_layer = tf.nn.sigmoid(tf.add(tf.matmul(x, w1), b1))
output_layer = tf.add(tf.matmul(hidden_layer, w2), b2)
```

Cost function and Optimizer:
The Cost is defined using the cross-entropy function and I'm using the Adam optimizer to minimize the cost.

```
# Cost funcition and optimizer
cost = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(
                        logits=output_layer, labels=y))
optimizer = tf.train.AdamOptimizer(
            learning_rate=learning_rate).minimize(cost)

# Define the Test model and accuracy
correct_prediction = tf.equal(tf.argmax(output_layer, 1),
                              tf.argmax(y, 1))
correct_prediction = tf.cast(correct_prediction, "float")
accuracy = tf.reduce_mean(correct_prediction)
```

Tensorflow session

```
# Launch the session
sess = tf.InteractiveSession()

# Initialize variables
init = tf.global_variables_initializer()

# Start session
sess.run(init)
# Accuracies arrays to create a plot
train_accuracies = []
validation_accuracies = []
epoc_iteration = []

# Run the session, save the accuracies
for epoch in range(training_epochs):
    batch_x, batch_y = mnist.train.next_batch(batch_size)
    if (epoch+1) < 100 or (epoch+1) % 100 == 0:
        train_ac = accuracy.eval({x: batch_x, y: batch_y})
        validation_ac = accuracy.eval(
                        {x: mnist.validation.images,
```

```
                        y: mnist.validation.labels})
        epoc_iteration.append(epoch+1)
        train_accuracies.append(train_ac)
        validation_accuracies.append(validation_ac)
    sess.run([optimizer, cost], feed_dict={x: batch_x,
                                           y: batch_y})
# Plot the training and validation accuracies
# Creates blank canvas
fig = plt.figure(figsize=(10,7))
axes1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])
axes2 = fig.add_axes([0.36, 0.25, 0.53, 0.5])

# Plot full graph
axes1.plot(epoc_iteration, train_accuracies,'-b',
                                label='Training')
axes1.plot(epoc_iteration, validation_accuracies,'-g',
                                label='Validation')
axes1.legend()
axes1.set_xlabel('Epoch')
axes1.set_ylabel('Accuracy')
axes1.set_title('Training and Validation accuracy')

# Plot zoom in graph
plt.ylim(ymax = 1.001, ymin = 0.95)
axes2.plot(epoc_iteration[198:],
           train_accuracies[198:],
           '-b', label='Training')
axes2.plot(epoc_iteration[198:],
           validation_accuracies[198:],
           '-g', label='Validation')
axes2.set_title('Zoom in');
```

```
# Print final accuracies
print("Validation Accuracy:",
        accuracy.eval({x: mnist.validation.images,
                       y: mnist.validation.labels}))
print("Test Accuracy:",
        accuracy.eval({x: mnist.test.images,
                       y: mnist.test.labels}))

#Validation Accuracy: 0.9732
#Test Accuracy: 0.9718
```

The final accuracy of this model is 97/100. The state-of-the-art model for this data set can give 99.7/100.

2 Proposed problem

2.1 Specification

Using tensorflow and neural networks with the example of training a model that is able to recognize digits, I will try to do the same with the first 10 letters of the alphabet (from "A" to "J").

The dataset I will use for this can be found here: `https://www.kaggle.com/lubaroli/notmnist/data`

Some articles that I have found useful:
`https://greydanus.github.io/2016/08/21/handwriting/`
`https://medium.com/@moshnoi2000/handwritting-recognition-using-tensorflow-aaf84fa9c587`
`https://niektemme.com/2016/02/21/tensorflow-handwriting/`
The first 2 articles present ideas on how to recognize whole sentences of handwriting, which although useful is not the idea of the project, I will try only to recognize particular characters, letters.
The third article presents, how a handwritten digits algorithm was made.

In this article, I found a lot of information about the NIST,MNIST and EMNIST datasets: `https://arxiv.org/pdf/1702.05373.pdf`

Presuming that the dataset is a good one and that there won't be a need for an image preparing function, I will have to make use of the os package from python to get the images, the the numpy package for converting them into vectors of 784 values (because the images are 28x28 pixels and 28*28=784), preferably a shuffling function for the images, cause otherwise the model wont be as accurate, the training will be done on batches, so a function for separating the data into batches.

The training will be done by calculating the cost function using the reduce_sum from the tensorflow package, and then from the same package the train.GradientDescentOptimizer function to train the model.

Of course, the data will have to be split into training and testing data.

2.2 Implementation

The data was manually taken using the os package, by running through each folder and assigning to each of the photos from a folder the corresponding label. While going through the images, 2 arrays are created, one of the images paths and one of labels, obviously these arrays are synchronized. Labels are the path to the image, because we need the path when reading the files from the batch function.

```
# returns 2 arrays
# the first contains the names of all images
# the second the corresponding labels
```

```
def getListOfImages(fromFolder):
    global folders
    global root

    allImagesArray = np.array([], dtype=np.str)
    allImagesLabelsArray = np.array([], dtype=np.str)

    for folder in folders:
        print("Loading Image Name of ", folder)
        currentFolder = root+fromFolder+"/"+folder+"/"
        imagesName = os.listdir(currentFolder)
        allImagesArray = np.append(allImagesArray,
            imagesName)
        #append all names of images (feature)
        print("Nr. of images: ", len(imagesName))
        for i in range(0, len(allImagesArray)):
            allImagesLabelsArray =
                np.append(allImagesLabelsArray,
                            currentAlphabetFolder)
            #append name of folder to each image (labels)
    return allImagesArray, allImagesLabelsArray
```

Just to be sure that the model is actually learning, these training examples we get from the array are shuffled, knowing that we have 300 examples for each letters, means a sum of 3000 images in total, so a shuffling of 100000 should be more than enough.

```
#returns randomized vector of (names+labels)
def shuffleImagesPath(imagesArray, labelsArray):
    print("Size to shuffle: ", len(imagesArray))
    for i in range(0, 100000):
        #random indexes
        r1 = randint(0, len(imagesArray)-1)
        r2 = randint(0, len(imagesArray)-1)
        #switch name and labels
        imagesArray[r1], imagesArray[r2] =
                imagesArray[r2], imagesArray[r1]
        labelsArray[r1], labelsArray[r2] =
                labelsArray[r2], labelsArray[r1]
    print("Shuffling done")
    return imagesArray, labelsArray
```

We shall create a neural network of the format 784-100-100-100-10, the first number consists of the features, while the last consists of the outputs (10 letters), the numbers between represent the hidden layers and their corresponding number of neurons.

```
# Network Parameters (784-100-100-100-10)
n_input = 784
hidden_layer_neurons = 100
hidden_layer2_neurons = 100
```

```
hidden_layer3_neurons = 100
n_classes = 10
```

Between all these layers we have an activation function which has a weight and a bias, these are arrays with size according to the previous and next layers and they are initialized with random values.

```
# Create weights and biases
# that will be used in the neural network
w1 = tf.Variable(
            tf.random_normal([n_input,
                              hidden_layer_neurons]))
w2 = tf.Variable(
            tf.random_normal([hidden_layer_neurons,
                              hidden_layer2_neurons]))
w3 = tf.Variable(
            tf.random_normal([hidden_layer2_neurons,
                              hidden_layer3_neurons]))
w4 = tf.Variable(
            tf.random_normal([hidden_layer3_neurons,
                              n_classes]))

b1 = tf.Variable(tf.random_normal([hidden_layer_neurons]))
b2 = tf.Variable(
        tf.random_normal([hidden_layer2_neurons]))
b3 = tf.Variable(
        tf.random_normal([hidden_layer3_neurons]))
b4 = tf.Variable(
        tf.random_normal([n_classes]))
```

As said above we have an activation function depending on the weight and bias, this activation function uses the information from the previous layer to calculate the information for the next layer.

```
# The multilayer perceptron model
hidden_layer = tf.nn.relu(
            tf.add(tf.matmul(x, w1),
                   b1))
hidden_layer2 = tf.nn.relu(
            tf.add(tf.matmul(hidden_layer, w2),
                   b2))
hidden_layer3 = tf.nn.relu(
            tf.add(tf.matmul(hidden_layer2, w3),
                   b3))
output_layer = tf.add(tf.matmul(hidden_layer3, w4), b4)
```

We calculate the cost function and use regularization on this cost function because neural networks are prone to overfitting, and then use the Adam optimizer to minimize the cost, to get the accuracy we look if the output is the same with the label.

```
# Cost funcition and optimizer
```

```
cost = tf.reduce_mean(
            tf.nn.softmax_cross_entropy_with_logits(
                        logits=output_layer,
                        labels=y))

regularizers = tf.nn.l2_loss(w1) +
                tf.nn.l2_loss(w2) +
                tf.nn.l2_loss(w3) +
                tf.nn.l2_loss(w4)
cost = tf.reduce_mean(cost + 0.05*regularizers);

trainer = tf.train.AdamOptimizer(learning_rate)
                    .minimize(cost)



# Define the Test model and accuracy
correct_prediction = tf.equal(
                            tf.argmax(output_layer, 1),
                            tf.argmax(y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction,
                            tf.float32))
```

Presuming that the images are not of the size we want, we shall resize the image to the desired size of 28x28, this matrix will be reshaped into a vector, this will represent the features, of size 784, this will be done for each image, this new vector will be added to a new array and the labels added to a label array, these 2 arrays will be again synchronized. To get the batch part, we will only append to the arrays until a counter gets to our desired batch size.

```
#get images in batches for training
def getBatchOfLetterImages(batchSize):
    global batchIndex
    global imagesPathArray
    global imagesLabelsArray

    # features are pixels, 784 features
    # labels are 10-vectors containing only one 1
    # which represents the letter
    # ex: first element is 1 for A, second for B...
    features = np.ndarray(shape=(0, 784), dtype=np.float32)
    labels = np.ndarray(shape=(0, 10), dtype=np.float32)

    with tf.Session() as sess:
        for i in range(startIndexOfBatch,
                len(imagesPathArray)):
            pathToImage = imagesLabelsArray[i]+
                imagesPathArray[i]
            lastIndexOfSlash = pathToImage.rfind("/")
            folder = pathToImage[lastIndexOfSlash - 1]
```

```
            imageContents = tf.read_file(str(pathToImage))
            image = tf.image.decode_png(imageContents,
                                        dtype=tf.uint8)
            resized_image =
                tf.image.resize_images(image, [28, 28])
            imarray = resized_image.eval()
            imarray = imarray.reshape(784)
            appendingImageArray = np.array([imarray],
                                           dtype=np.float32)
            appendingNumberLabel =
                    np.array([getNumber(folder)],
                             dtype=np.float32)
            labels =
                    np.append(labels,
                              appendingNumberLabel,
                              axis=0)
            features =
                    np.append(features,
                              appendingImageArray,
                              axis=0)
            batchIndex=batchIndex+1
            if(len(labels) >= batchSize):
                return labels, features
```

The getting of test arrays, works in the same idea as above but without the batch part.

```
def getTestArrays(testImages, testLabels):
    dataset = np.ndarray(shape=(0, 784), dtype=np.float32)
    labels = np.ndarray(shape=(0, 10), dtype=np.float32)
    with tf.Session() as sess:
        for i in range(0, len(testImages)):
            pathToImage = testLabels[i]+testImages[i]
            lastIndexOfSlash = pathToImage.rfind("/")
            folder = pathToImage[lastIndexOfSlash - 1]

            imageContents = tf.read_file(str(pathToImage))
            image = tf.image.decode_png(imageContents,
                                        dtype=tf.uint8)
            resized_image = tf.image.resize_images(image,
                                                   [28, 28])
            imarray = resized_image.eval()
            imarray = imarray.reshape(784)
            appendingImageArray = np.array([imarray],
                                           dtype=np.float32)
            appendingNumberLabel = np.array(
                                   [getNumber(folder)],
                                   dtype=np.float32)
            labels = np.append(labels,
```

```
                                    appendingNumberLabel,
                                    axis=0)
            dataset = np.append(dataset,
                                    appendingImageArray,
                                    axis=0)
        return dataset, labels
```

As said in the previous parts of this document, in tensorflow everything is done in the session, here we train the model on the same data several times (epochs), throughout each epoch the model gets all the training examples in batches and it learns.

```
with tf.Session() as session:
    session.run(tf.global_variables_initializer())
    for i in range(0, epochs):
        batchIndex=0
        for j in range(0, nrBatches):
            batchY, batchX =
                    getBatchOfLetterImages(batch_size)
            opt = session.run(trainer,
                        feed_dict={x: batchX, y: batchY})
            loss, acc = session.run([cost, accuracy],
                        feed_dict={x: batchX, y: batchY})
            print("Iteration: " + str(j+1) + ", Loss= " +
                        "{:.6f}".format(loss) +
                        ", Training Accuracy= " +
                        "{:.5f}".format(acc))
        print("Epoch: " + str(i+1) +
                ", Loss= " + "{:.6f}".format(loss) +
                ", Training Accuracy= " +
                "{:.5f}".format(acc))
    print("Test accuracy: ", accuracy.eval(
            feed_dict={x: testData, y: testDataLabel}))
```

2.3 Documentation of your solution

The data we have are images that have a 28x28 format, these images are transformed into vectors of 28*28=784 numbers which are the values of the pixels. These 784 numbers are the features of one training example. The labels are created after the one hot model, which means that for 10 letters from A to J, we will have 10 vectors, each vectors contains only one 1 and nine 0s, when the first element of the vector is 1, it represents A, when the second, it represents B and so on.

After training the model on 300 training examples for each letter (a total of 3000 images) through 5 epochs the training accuracy goes up to about 80-85%, the testing was done on 1000 images (100 of each letter) and it got an accuracy of 70%, these results were possible only by meddling with the learning rate and regularization parameter.

3 References

1. TensorFlow installation guide: https://www.tensorflow.org/install/
2. TensorFlow training at Edureka:https://www.edureka.co/ai-deep-learning-with-tensorflow
3. Hands-On Machine Learning with Scikit-Learn and Tensorflow