Cognitive Computer Vision: Exercise 2

1 Variables

When you run operations, all of the results computed are only stored until the computation is finished. The exception to this rule are tensors defined via `tf.Variable`. A variable allows you to define data that persists over many session run calls. Variables are useful for storing model parameters, or bookkeeping such as remembering the number of iterations.

- `tf.Variable` is not an op. Recall: ops have lowercase names.
- Assign a new value to the variable with `assign()` or a related op.
- A variable must be initialized before use!

Run and study the following code. Make sure you understand what each line does.

```python
import tensorflow as tf

x = tf.ones(1, name="x", dtype=tf.int32)
x_plus_one = x+1

xvar = tf.Variable([1], name="x_variable")
xvar_plus_one = xvar.assign(xvar + 1)

sess = tf.Session()
sess.run( xvar.initializer )
for i in range (5):
  print(sess.run( x_plus_one ))
  print(sess.run( xvar_plus_one ))
  print('')
```

Listing 1: Using a variable

Write a Tensorflow program that calculates the running average of a sequence $(x_1, x_2, \ldots)$ of real numbers provided by the user one number at a time. The running average $\hat{x}_n$ for numbers up to and including x_n is be calculated recursively by initializing $\hat{x}_0 = 0$, then for any $n \geq 1$:

$$\hat{x}_n = \frac{x_n}{n} + \frac{(n-1) \cdot \hat{x}_{n-1}}{n}. \tag{1}$$

Hints:

- Use a scalar placeholder for feeding the data x_n. Use one `tf.Variable` to remember n, another to remember $\hat{x}_n$; both should have initial value 0. Use `tf.float32` datatype[1].
- Always increment n before computing the new value of $\hat{x}_n$. There are two ways to do this.
  1. You can make two calls to `session.run()` to first increment n, then to update $\hat{x}_n$, or

[1]If you want to use an integer datatype for n, explicitly `tf.cast` it to floating point when dividing with it.

2. use a *control dependency*, which enforces that some ops should be run before others:

```
increment_n = n.assign_add (1.0)
with tf.control_dependencies([increment_n]):
  # here: define other ops that depend on first running op
   increment_n , for example:
  update_avg = x_avg.assign( ... )
```

Listing 2: Operation `update_avg` depends on first running ops in the list `[increment_n]`.

- Verify your program runs correctly. For example, for input sequence $(0, 2, 4, 6, 8)$ the running averages output should be $(0, 1, 2, 3, 4)$.

2 Fully connected layer

Given input $x \in \mathbb{R}^m$, a fully connected or dense neural network layer with k output neurons returns $y \in \mathbb{R}^k$,

$$y = h(Wx + b), \tag{2}$$

where $W \in \mathbb{R}^{k \times m}$ is the matrix of weights, $b \in \mathbb{R}^k$ is the bias vector, and $h : \mathbb{R} \to \mathbb{R}$ is the activation function. A dense layer with k output neurons and a linear activation function $h(x) = x$ can be defined by

```
y = tf.layers.dense(x, k, activation=None , use_bias=True , name="
   my_dense_layer")
```

See `https://www.tensorflow.org/api_docs/python/tf/layers/dense` for the full documentation.

Write a program that calculates y by Eq. (2) when W is a 2-by-2 matrix, x and b are 2-element vectors, and h is the rectified linear unit (ReLU) activation function

$$h(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{otherwise} \end{cases}, \tag{3}$$

see `https://www.tensorflow.org/api_guides/python/nn#Activation_Functions`.

- `tf.layers.dense` creates variables for the bias b and weights W. As the variables are created "behind the scenes", we cannot directly run their initializers. Instead use the convenience operation `tf.global_variables_initializer()`:

```
# Initialize all variables before running other ops
sess.run( tf.global_variables_initializer() )
```

Listing 3: `tf.global_variables_initializer()` will initialize all variables in the graph.

- `tf.layers.dense` is optimized to operate on many inputs at once. Thus, it expects its input x to be of size (N, m) where N is the number of inputs, and m is the dimension of each input, as in Eq. (2) above.

- Find how to define the layer to initialize both W and b to some known values, for example

$$A = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, b = \begin{bmatrix} 1 \\ 1 \end{bmatrix}.$$

  See `https://www.tensorflow.org/api_docs/python/tf/initializers`. Verify you get the correct result: for example, for $x = \begin{bmatrix} 1 & -2 \end{bmatrix}^T$, $Wx + b = \begin{bmatrix} 2 & -1 \end{bmatrix}^T$ and thus $h(Wx + b) = \begin{bmatrix} 2 & 0 \end{bmatrix}^T$.

- Look at TensorBoard and locate the variables and see what operations are included in the graph.

3 Linear regression: optimization and logging data

Linear regression means modelling the linear relationship between an input variable x and an output variable y. We are given n samples of the input variable, $(x_1, x_2, \ldots, x_n)$, and n samples of the output variable; $(y_1, y_2, \ldots, y_n)$. This data set is depicted by the blue points in Figure 1. We want to find the best linear model of the form $\hat{y} = \hat{a}x + \hat{b}$ that explains the dependence of the output on the input. By best, we mean here the *least squares* solution, i.e., the choice for the model parameters $\hat{a}$ and $\hat{b}$ that minimizes the sum-of-squares error

$$\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 = \sum_{i=1}^{n} \left((\hat{a}x_i + \hat{b}) - y_i\right)^2.$$

The line $\hat{y} = \hat{a}x + \hat{b}$ that minimizes the error above is depicted by the red line in Figure 1.

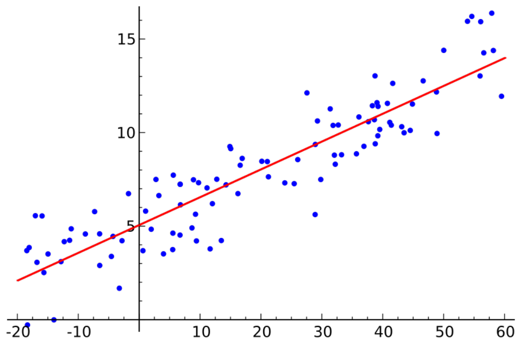

Figure 1: The red line is the least-squares solution to linear regression on the blue data points.

Notice that the linear model equation exactly matches what the dense (fully-connected) layer with a linear activation function if all inputs are scalars. We will see if we can recover the same parameters $\hat{a}$ and $\hat{b}$ by solving the linear regression problem using Tensorflow.

Start with the code below that creates a synthetic dataset from known parameters and visualizes it. It also calculates the well-known closed-form solution to the problem by

$$\begin{bmatrix}\hat{a} & \hat{b}\end{bmatrix}^T = (X^T X)^{-1} X^T \vec{y},$$

where

$$X = \begin{bmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{bmatrix}, \vec{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

```
import numpy as np
import matplotlib.pyplot as plt
a = 1.1
b = 0.2
n = 128 # number of points in dataset

# Dataset : added normally distributed noise to make it interesting
x = np.random.uniform(low=-1.0, high=1.0, size=n)
y = a * x + b + np.random.normal(loc=0.0, scale=0.4, size=n)

# The least squares solution calculated explicitly
X = np.vstack([x, np.ones(len(x))]).T
a_hat, b_hat = np.linalg.lstsq(X, y)[0]
print(a_hat, b_hat)
```

```

# Values at linearly spaced x for plotting
xg = np.linspace(-1.0, 1.0, num=20)
yg = a * xg + b

plt.plot(x, y, 'bo')
plt.plot(xg, yg, 'r')
plt.ylabel('y')
plt.xlabel('x')
plt.show()
```

Listing 4: Creating a dataset for linear regression.

- Create placeholders `xp` and `yp` for the dataset, with shape `[n,1]` and datatype `tf.float32`.
- Create a dense layer to output a prediction $\hat{y}$. Use default initializer for weight and bias.
- Define a loss function using `loss = tf.losses.mean_squared_error( ... )` to compare $\hat{y}$ to the placeholder input y.
- Use a gradient descent optimizer to minimize the loss function:

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.1)
minimize_op = optimizer.minimize(loss)
```

Listing 5: Setting up an optimizer to minimize a loss function.

- Fill in missing parts marked by `TODO` to the following code snippet to run the optimization:

```
# Get names of trainable tf.Variables in the graph
variables_names = [v.name for v in tf.trainable_variables()]
# Set up logging of the loss value
loss_summary = tf.summary.scalar(name="loss", tensor=loss)

with tf.Session() as sess:
  # TODO: Create FileWriter to save graph!
  # TODO: initialize all variables here!
  for k in range(100):
    # TODO: fill in the empty feed_dict!
    _, l = sess.run([minimize_op, loss_summary], feed_dict={})
    # TODO: use FileWriter.add_summary() to log l. Use k as global step!

  # After training, fetch trained variables and print values.
  trained_values = sess.run(variables_names)
  for k, v in zip(variables_names, trained_values):
      print("Variable: " + k + ' value: ' + str(v))
```

Listing 6: Fill the missing parts for linear regression.

Can you find parameters that are close to those the explicit least squares solution? Look in TensorBoard to see a visualization of the loss per iteration. Notice that if your loss increases instead of decreases, the learning rate may be too high.