Intro to NLP and Deep Learning - 236605

Tutorial 4 – Tensorflow Introduction

- Intro
- Tensors
- Tensorflow core - constants, placeholders, variables
- Implementation of Feed-forward network

Tomer Golany – tomer.golany@gmail.com
Introduction
TensorFlow - Introduction

- TensorFlow is an open source software library for numerical computation using data flow graphs.

- **Nodes** in the graph represent mathematical operations, while the graph **edges** represent the multidimensional data arrays (tensors) communicated between them.

- The flexible architecture allows you to deploy computation to one or more **CPUs** or **GPUs** in a desktop, server, or mobile device with a single API.

- TensorFlow was originally developed by researchers and engineers working on the Google Brain Team within Google's Machine Intelligence research organization for the purposes of conducting machine learning and deep neural networks research.
Tensors
The central unit of data in TensorFlow is the **tensor**.

A tensor consists of a set of primitive values shaped into an array of any number of dimensions.

A tensor’s rank is its number of dimensions. Here are some examples of tensors:

- A rank 0 tensor; a scalar with shape []
- A rank 1 tensor; a vector with shape [3]
- A rank 2 tensor; a matrix with shape [2, 3]
- A rank 3 tensor with shape [2, 1, 3]

When writing a TensorFlow program, the main object you manipulate and pass around is the **tf.Tensor**.

A tf.Tensor object represents a partially defined computation that will eventually produce a value.

TensorFlow programs work by first building a graph of tf.Tensor objects and then by running parts of this graph to achieve the desired results.
A tf.Tensor has the following properties:

- Each element in the Tensor has the same data type.
- The shape (that is, the number of dimensions it has and the size of each dimension) might be only partially known.
- Main type of tensors:
  - tf.Variable
  - tf.Constant
  - tf.Placeholder
- With the exception of tf.Variable, the value of a tensor is immutable, which means that in the context of a single execution tensors only have a single value.
- However, evaluating the same tensor twice can return different values; for example that tensor can be the result of generating a random number.
You might think of TensorFlow Core programs as consisting of two discrete sections:

- **Building the computational graph.**
- **Running** the computational graph.

A **computational graph** is a series of TensorFlow operations arranged into a graph of nodes.

- Each node takes zero or more tensors as inputs and produces a tensor as an output.

**Constant tensors:**

- TensorFlow constants take no inputs, and it outputs a value it stores internally. We can create two floating point Tensors node1 and node2 as follows:

```python
node1 = tf.constant(3.0, dtype=tf.float32)
node2 = tf.constant(4.0) # also tf.float32 implicitly
print(node1, node2)
```

- Output:

```
Tensor("Const:0", shape=(), dtype=float32) Tensor("Const_1:0", shape=(), dtype=float32)
```
The Computational Graph - Session

- Printing the nodes does not output the values 3.0 and 4.0 as we expected.
- Instead, they are nodes that, **when evaluated**, would produce 3.0 and 4.0, respectively.
- To actually evaluate the nodes, we must run the computational graph within a **session**.

- A **session** encapsulates the control and state of the TensorFlow runtime.

- A Session object can run a computational graph with the **method run**

- Lets run the simple computational graph to evaluate the two constant nodes:

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

- Output: `[3.0, 4.0]`
The Computational Graph - Operations

- We can build more complicated computations by combining Tensor nodes with **operations**.
- Operations are also nodes - they take tensors as input, and produce tensors as output.

- For example, we can add our two constant nodes and produce a new graph as follows:

```python
node3 = tf.add(node1, node2)
print("node3:", node3)
print("sess.run(node3):", sess.run(node3))
```

- Output:

```
node3: Tensor("Add:0", shape=(), dtype=float32)
sess.run(node3): 7.0
```
The graph last slide is not especially interesting because it always produces a constant result.

A graph can be parameterized to accept external inputs, known as **placeholders**.

A **placeholder** is a promise to provide a value later.

For example, we can add our two constant nodes and produce a new graph as follows:

```python
a = tf.placeholder(tf.float32)
b = tf.placeholder(tf.float32)
adder_node = a + b  # + provides a shortcut for tf.add(a, b)
```

The preceding three lines are a bit like a function or a lambda in which we define two input parameters (a and b) and then an operation on them.

We can evaluate this graph with multiple inputs by using the **feed_dict** argument to the run method.
Placeholders

- Running the computational graph with different inputs:

```python
print(sess.run(adder_node, {a: 3, b: 4.5})))
print(sess.run(adder_node, {a: [1, 3], b: [2, 4]}))
```

- Output:

```python
7.5
[3.7]
```

- We can make the computational graph more complex by adding another operation. For example:

```python
add_and_triple = adder_node * 3.
print(sess.run(add_and_triple, {a: 3, b: 4.5}))
```

Output: **22.5**
In machine learning we will typically want a model that can take arbitrary inputs, such as the one above:

- To make the model trainable, we need to be able to modify the graph to get new outputs with the same input.
- **Variables** allow us to add trainable parameters to a graph. They are constructed with a type and initial value.

```python
W = tf.Variable([.3], dtype=tf.float32)
b = tf.Variable([-3], dtype=tf.float32)
x = tf.placeholder(tf.float32)
linear_model = W*x + b
```
Variables - Initialization

To initialize all the variables in a TensorFlow program, you must explicitly call a special operation as follows:

```python
init = tf.global_variables_initializer()
sess.run(init)
```

Until we call `sess.run`, the variables are uninitialized.

Since `x` is a **placeholder**, we can evaluate `linear_model` for several values of `x` simultaneously as follows:

```python
print(sess.run(linear_model, {x: [1, 2, 3, 4]}))
```

Output:

```python
[ 0. 0.30000001 0.60000002 0.90000004]
```
Training Linear model

To evaluate the model on training data, we need a $y$ placeholder to provide the desired values, and we need to write a loss function.

We'll use a standard loss model for linear regression, the squares errors function.

We will use `tf.square` method to square the error.

We will sum all the squared error using `tf.reduce_sum` function.

```python
y = tf.placeholder(tf.float32)
squared_deltas = tf.square(linear_model - y)
loss = tf.reduce_sum(squared_deltas)

print(sess.run(loss, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]}))

Output: 23.66
```
Training Linear model

- TensorFlow provides optimizers that slowly change each variable in order to minimize the loss function.

- The simplest optimizer is gradient descent.

```python
optimizer = tf.train.GradientDescentOptimizer(0.01)

train = optimizer.minimize(loss)

for i in range(1000):
    sess.run(train, {x: [1, 2, 3, 4], y: [0, -1, -2, -3]})

print(sess.run([W, b]))

Output:
[ array([-0.9999969], dtype=float32), array([ 0.99999082], dtype=float32) ]
```
Implementation of Feed Forward NN
We will now implement a **Fully connected** network with 1 hidden layer. (same as we saw in tutorial #2)
Defining the input variables

- First step will be to define the input variables of the network.
- The dimension of each input sample to the network is 2
- we will add a third dimension which will always hold the value 1. this dimension will be multiplied with the bias.

```python
# Define input Placeholders:
x_dim = 2
y_dim = 2
X = tf.placeholder(tf.float32, shape=[None, x_dim + 1])
y = tf.placeholder(tf.float32, shape=[None, y_dim])
```

- We want to feed a batch of examples to the network each time, therefore we define X and Y as a matrix with dimensions: [None, 2] .
- None keyword means that the input variable can accept any number of dimensions.
- So we defined the X to be a matrix of BATCH_SIZE X 2 - each row. In the matrix is one sample.
- Each label Y also has 2 dimensions.
We saw last tutorial (in the skip-gram model) that each layer in a fully connected network can be represented with a weights matrix. We will define these matrices for the hidden and output layer.

```python
# Weight initializations

# Layer 1:
# hidden_layer_w_matrix = tf.Variable(tf.random_normal((x_dim + 1, num_of_neurons_at_hidden_layer), stddev=0.1))
hidden_layer_w_matrix = tf.Variable([[0.15, 0.25], [0.2, 0.3], [0.35, 0.35]], dtype=tf.float32)

# Layer 2:
# final_layer_w_matrix = tf.Variable(tf.random_normal((num_of_neurons_at_hidden_layer + 1, y_dim), stddev=0.1))
final_layer_w_matrix = tf.Variable([[0.4, 0.5], [0.45, 0.55], [0.6, 0.6]], dtype=tf.float32)
```
Forward Pass

To define the multiplication between the weights and the input samples we will use the method \texttt{tf.matmul}:

- \texttt{\textbf{matmul}(a, b, transpose\_a=False, transpose\_b=False, adjoint\_a=False, adjoint\_b=False, a\_is\_sparse=False, b\_is\_sparse=False, name=None)}

  - Multiplies matrix a by matrix b, producing \( a \times b \).

  - **a**: Tensor of type float16, float32, float64, int32, complex64, complex128 and rank \( > 1 \).

  - **b**: Tensor with same type and rank as a.

\[
\text{hidden\_layer\_output\_before\_activation} = \text{tf.matmul}(X, \text{hidden\_layer\_w\_matrix})
\]
Tensorflow has implementation for all activation functions under `tf.nn` api.

We will use `tf.nn.sigmoid` to activate sigmoid on the output of the hidden layer.

```python
hidden_layer_output_after_activation = tf.nn.sigmoid(hidden_layer_output_before_activation)
```

- Same operations will be done on the output layer:

```python
# Add bias for final layer:
hidden_layer_output_after_activation = tf.concat([hidden_layer_output_after_activation, np.ones((batch_size, 1), float)], 1)

final_layer_output_before_activation = tf.matmul(hidden_layer_output_after_activation, final_layer_w_matrix)

softmax_activation_on_final_layer = tf.nn.softmax(final_layer_output_before_activation)
```
Calculate Cost

cost_per_instance = -tf.reduce_sum((y) * tf.log(softmax_activation_on_final_layer), reduction_indices=[1])

total_cost_per_batch = tf.reduce_mean(cost_per_instance)

# cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(labels=y, logits=prediction))
predict = tf.argmax(softmax_activation_on_final_layer, axis=1)
Backpropagation and Training:

```python
updates = tf.train.GradientDescentOptimizer(0.01).minimize(total_cost_per_batch)

with sess as tf.Session():
  sess.run(updates, feed_dict={X: x_input, y: y_tags})

train_accuracy = np.mean(np.argmax(y_tags, axis=1) ==
  sess.run(predict, feed_dict={X: x_input, y: y_tags}))
```