TensorFlow

HW 4/5

Project Plans

TensorFlow Operations

- Node in the dataflow graph
- In general, perform some service or computation
- In many cases, they produce a result
  - Can have any number of channels (0, 1, …). But: in most cases, there are zero channels or one channel as output
- Results are edges in the graph

Operations and Tensors

```
x = tf.constant(ins_training, dtype=tf.float32, name = "x")
```

- ins_training: numpy array in this case
- name: name of the node

This line creates:

- Operation node
- Output tensor, which is returned and stored in variable x

Tensors

Objects that:

- Contain information about how to evaluate them (i.e. a reference to the associated node)
- Have a value (a mathematical tensor), which can be obtained using the eval() method

Variable Nodes

```
w = tf.Variable(tf.random_uniform([n,1], -1.0,
      1.0), name = "w")
```

- Creates a “variable” type node
- Creates the tensor that evaluates to the value of the variable

```
a = w.eval()
```

- Evaluate the tensor
- In this case, a is a numpy array

General Operation Nodes

```
error = y_pred - y
```

- Create a general operation node (a name will be assigned)
- Creates an output tensor, which is assigned to variable error_training

- The TensorFlow class provides a set of functions that will perform various kinds of mathematical operations (and they can be named):

```
error = tf.subtract(y_pred, y, name="error")
```

Placeholders

```
n = number of samples
x = tf.placeholder(dtype=tf.float32,
                   shape=(None, n), name="x")
```

- Create a node that has a (partially) defined shape and whose value will be defined later
- x is the corresponding Tensor

Placeholders

- All evaluation / running involves touching a subset of the graph
- If this subgraph involves a placeholder, then its value must be defined at time of evaluation / running

```
feed_dict_validation = {x: ins_validation,
                        y: outs_validation}
f = fvaf.eval(feed_dict = feed_dict_validation)
```

Not all Operation Nodes Produce a Tensor

```
training_op = tf.assign(w,
                        w - alpha * gradients)
```

- Return value is a node (not a Tensor!)
- Nodes can be “run” but not eval’d:

```
sess.run(training_op)
```

Graph Evaluation

Evaluating or running a node is generally recursive:

- Evaluation in the graph will stop at a variable, constant and placeholder nodes
- But: general operations will recursively evaluate their input Tensors before performing their operation
- This means that separate calls to eval() or run() will cause reevaluation …

Graph Evaluation

Evaluating multiple variables:

```
[fvaf_training, mse_training] =
    sess.run([fvaf, mse], feed_dict=feed_dict)
```

Name Scopes

Like name spaces in general, this is a convenient way to organize the nodes within your graph

```
with tf.name_scope(“my_network”):
    x = tf.placeholder(dtype=tf.float32,
                       shape=(None, n), name="x")
```

x variable still refers to this Tensor, but its name is now “my_network/x”. There could also be a “your_network/x”

Accessing Tensors by Name

```
# Default graph
g = tf.get_default_graph()

# Often passed in as a function parameter
namespace = “my_network”

# Get handles to the input/desired output
x = g.get_tensor_by_name(namespace + "/x:0")
```

- namespace + “/x” is the node, “0” is the output channel
- x variable is the corresponding Tensor

Accessing Operations by Name

```
# Default graph
g = tf.get_default_graph()

# Often passed in as a function parameter
namespace = “my_network”

# Get handles to the input/desired output
training_op = g.get_operation_by_name(namespace +
                                      "/training_op")
```

- training_op is the operation

Growing Bigger Graphs

- When you are first playing with small graphs, it is easy to be sloppy about your global variables
- This becomes unwieldy as things get bigger

Growing Bigger Graphs

Best (?) practices for larger graphs:

- Heavy use of name scopes
- Encapsulate graph construction and use within functions
  - Mostly use local variables (if not all the time)
  - Rely on find_operation_by_name() and find_tensor_by_name() within functions to access the references that you need

Deep Learning

- So far, we have relied on hand-selecting the features that we use as input to our models
- With the right set of features, our models can even be linear!
- But – it is not always clear which features we should use.
  - Especially the case when we have so many input variables

Deep Learning

Cascades of multiple neural layers

- Early layers: enables us to “learn” new non-linear features that can then be used as input to later stages of the network
- In some cases, these first stages of features are useful in other contexts
  - Enable transfer of knowledge

Challenges

- Deciding on structure
  - How many layers?
  - How many neurons in each layer
- Gradient problem
  - Vanishing gradient
  - Exploding gradient
- Many parameters:
  - Over-fitting

Dealing with these challenges requires empirical work

Structure

- The learning algorithm will make use of all of its degrees of freedom to re-represent its input information at each stage
- If a layer has an equal number or more neurons than the previous layer, then a simple solution is to essentially “copy” the prior layer’s state
- This means that the algorithm has no “incentive” to find an alternative representation

Structure

This means that the algorithm has no “incentive” to find an alternative representation

- By reducing the number of neurons from one layer to another, we force the learning algorithm to compress the number of degrees of freedom
- The available degrees of freedom must be used to capture the most variance in the input data, while preserving the information that we need to explain the variance in the output

Structure

- Exactly what the structure should be is dependent on the problem you are trying to solve
- Many different bits of wisdom available in the literature
- What it comes down to: be ready to embark on an empirical process
  - Start simple and expand from there

Gradient Problem

Deeper networks -> more opportunity for the gradient to go to zero

- Initialization of weight parameters: select so that most neurons are not saturated to begin with
- Choice is dependent on the type of nonlinearity

Gradient Problem

Use nonlinearities that (almost) always have a non-zero gradient

from keras.layers import LeakyReLU

- LeakyReLU(beta): beta is the slope of the activation function below zero
- Returns a function that return takes a Tensor as input and that returns a Tensor

Gradient Problem

Batch Normalization:

- Extra step “between” layers
- Given a training data set, center and scale the inputs to the next layer so that they tend not to lead to saturation in the next layer