11-695: AI Engineering
Coding a Neural Network

Spring 2019

Outline

Recap: Tensorflow

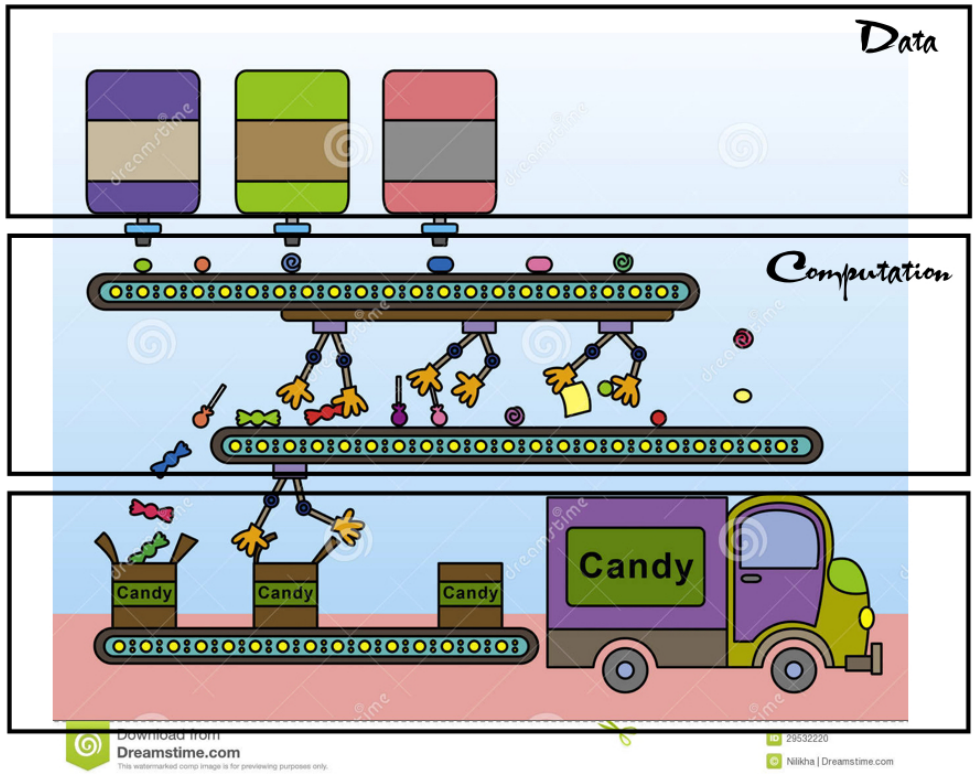
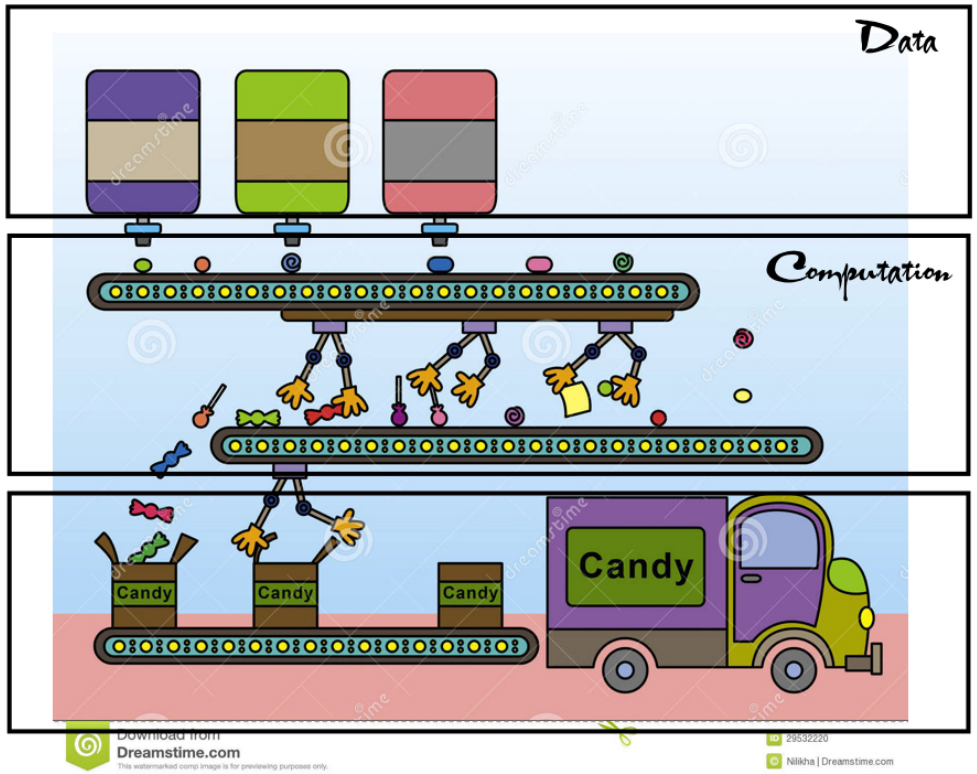


Image credit: Dreamstime.com

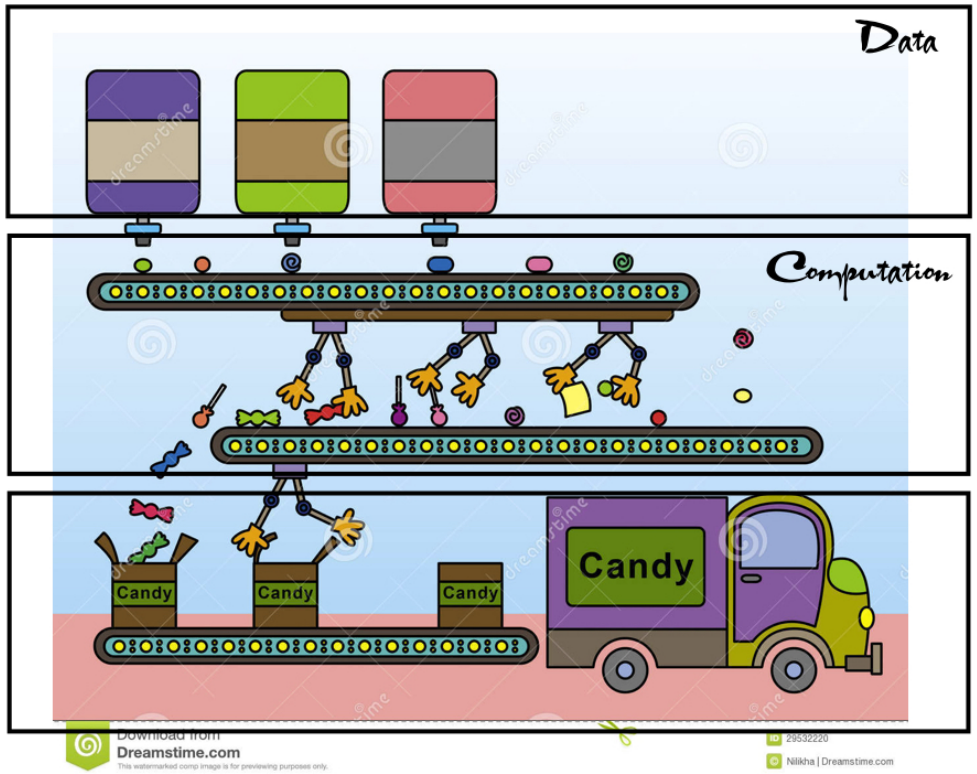
Recap: Tensorflow



- Distinguish between computational graph and data

Image credit: Dreamstime.com

Recap: Tensorflow



- Distinguish between computational graph and data
- Understand the role of a Session

Image credit: Dreamstime.com

Outline

General TF Structure

Carnegie Mellon

tf_graph_basis.py

```
1  import tensorflow as tf
2
3  def build_dataset():
4    # process data in a "TF-friendly" way
5
6  def build_tf_graph():
7    # create variables, ops
8
9  def main(_args):
10   g = tf.Graph()
11   with g.as_default():
12     ops_dict = build_tf_graph()
13     with tf.Session() as sess:
14       output = sess.run([train_op, ...])  # execute the op "z" in the graph
```

- Computational Graph:
  - nodes or *ops*
  - edges are *tensors*

General TF Structure

Carnegie Mellon

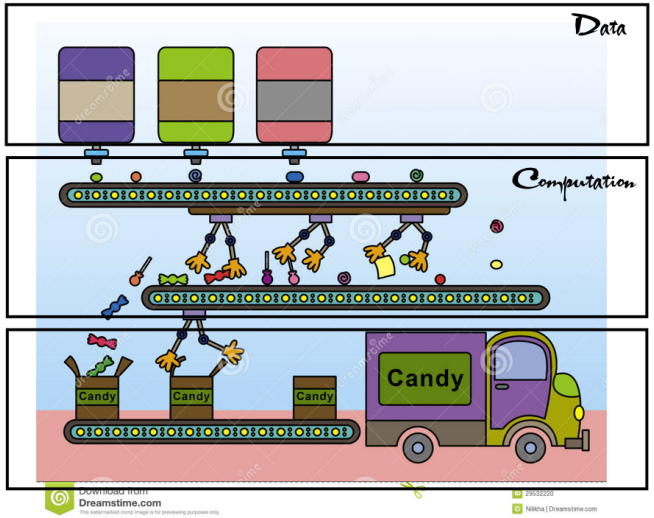
tf_graph_basis.py

```
1  import tensorflow as tf
2
3  def build_dataset():
4    # process data in a "TF-friendly" way
5
6  def build_tf_graph():
7    # create variables, ops
8
9  def main(_args):
10   g = tf.Graph()
11   with g.as_default():
12     ops_dict = build_tf_graph()
13     with tf.Session() as sess:
14       output = sess.run([train_op, ...])  # execute the op "z" in the graph
```

- Execution (or Evaluation): `sess.run([a, a, b, c])`
  - `a`, `b`, `c` *and their parents* will be run
  - Two `a`s will be run *once*
- Using `tf.Estimator` also has the same structure in general

Outline

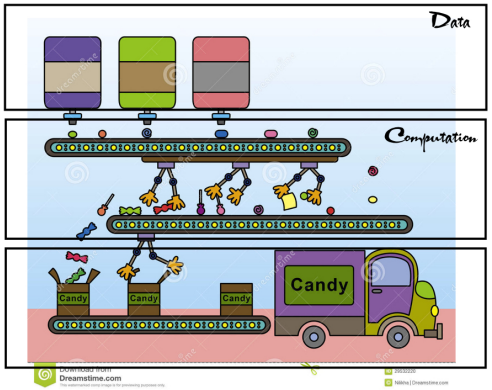
General Steps for Data Processing



- Parse data
- Pre-process data for specific tasks
- Augment data if needed (generally a good idea)
- And *make data ready for Tensorflow to consume.*
  - This should be thought first!

Interfacing Data with Graph

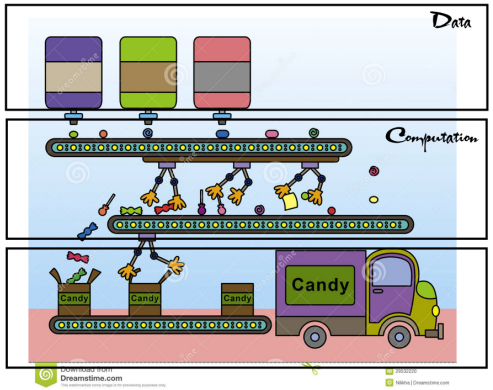
Carnegie Mellon



- *i.e.* Convert data to Tensors
  - Graph cannot consume data directly
  - Graph only contains ops, Tensor is an op
- 3 options
  - Late interfacing: parse, preprocess, augment and put them in a `tf.placeholder` which wraps data to tensors

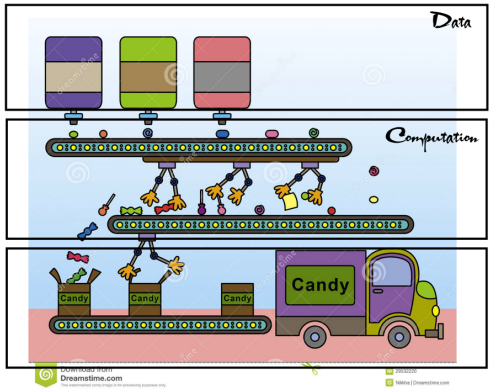
Interfacing Data with Graph

Carnegie Mellon



- *i.e.* Convert data to Tensors
  - Graph cannot consume data directly
  - Graph only contains ops, Tensor is an op
- 3 options
  - Late interfacing: parse, preprocess, augment and put them in a `tf.placeholder` which wraps data to tensors
  - (Very) Early interfacing: "tensorize" all those steps with `tf.Data`

Interfacing Data with Graph



- *i.e.* Convert data to Tensors
  - Graph cannot consume data directly
  - Graph only contains ops, Tensor is an op
- 3 options
  - Late interfacing: parse, preprocess, augment and put them in a `tf.placeholder` which wraps data to tensors
  - (Very) Early interfacing: "tensorize" all those steps with `tf.Data`
  - Mixture of the two (see HW1 - ImageNet)

Late Interfacing with tf.placeholder

Carnegie Mellon

tf_placeholder.py

```
1  def build_dataset():
2    # parse, pre-process, augment and importantly, do the batching properly yourself
3    return (batch_numpy_data, batch_numpy_labels)
4
5  def build_tf_graph():
6    batched_data = tf.placeholder(dtype=tf.float32, shape=[32, 128, 128, 3],
7                                  name='data_tensor')
8    batched_labels = tf.placeholder(dtype=tf.int32, shape=[32, ], name='label_tensor')
9    # create variables, ops
10
11 def main(_args):
12   g = tf.Graph()
13   with g.as_default():
14     ops_dict = build_tf_graph()
15     with tf.Session() as sess:
16         for epoch in range(n_epochs):
17             for i in range(n_batch_per_epoch):
18                 (batched_numpy_data, batched_numpy_labels) = build_dataset()
19                 feed_dict = {batched_data: batched_numpy_data,
20                              batched_labels: batched_numpy_labels}
21
22                 output, ... = sess.run([train_op, ...], feed_dict=feed_dict)
23                 # do smth with outputs
```

Late Interfacing with `tf.placeholder`

Carnegie Mellon

short_tf_placeholder.py

```
1  def build_dataset():
2    # and importantly, do the batching properly yourself
3    return (batch_numpy_data, batch_numpy_labels)
4
5  def main(_args):
6      # ... omitted ...
7      feed_dict = {batched_data: batched_numpy_data, batched_labels: batched_numpy_labels}
8      output = sess.run([train_op, ...], feed_dict=feed_dict)
```

- You have to deal with Batching yourself
  - Decide between drawing a batch *with* or *without* replacement
- `Feed_dict` is the actual interface
  - Must define the `tf.placeholder` op with the correct shapes and data types!
- Must do for all non-tensor data that need consuming by the graph!
- Might be gone with Eager Execution, still be a memory-efficient implementation for graph. But laboring?

Early Interfacing with `tf.data.Dataset`[1]

Carnegie Mellon

- Create a Dataset object from:
  - numpy data (you have done parsing the data into Numpy)
  - file paths to the data (you have just only done collecting paths)
- In a multiple ways:
  - `tf.data.Dataset.from_tensors`
  - `tf.data.Dataset.from_tensor_slices`
  - `tf.data.Dataset.from_generator` - memory efficiency

[1] `https://www.tensorflow.org/guide/datasets`

Early Interfacing with `tf.data.Dataset`[1]

Carnegie Mellon

- Create a Dataset object from:
  - numpy data (you have done parsing the data into Numpy)
  - file paths to the data (you have just only done collecting paths)
- In a multiple ways:
  - `tf.data.Dataset.from_tensors`
  - `tf.data.Dataset.from_tensor_slices`
  - `tf.data.Dataset.from_generator` - memory efficiency
- If you are sure about data format, use a child class:
  - `tf.data.TextLineDataset`
  - `tf.data.FixedLengthRecordDataset`
  - `tf.data.TFRecordDataset`
  - `tf.data.ParallelInterleaveDataset`
  - Many more experimentals: `Random`, `CSV`, `SQL`, `Kafka`, `SequenceFile`, `Kinesis`, `MachineFiles`, `LMDB`

[1] `https://www.tensorflow.org/guide/datasets`

tf.data.Dataset Data Preprocessing[2]

Carnegie Mellon

tf_dataset_preprocessing.py

```
1  def preprocessing(data_tensor, label_tensor):
2    # do something with tf libraries
3    # tf.pad, tf.random_crop, tf.random_brightness, ...
4
5  def build_dataset():
6     dataset = tf.data.Dataset.from_tensor_slices((img_paths_tensor, label_paths_tensor))
7     dataset = dataset.map(parse_data,  num_parallel_calls=10)
8     dataset = dataset.map(preprocessing,  num_parallel_calls=10)
```

- Preprocessing via transformation
  - `tf.data.Dataset.map`
  - `tf.data.Dataset.flatmap`
  - `tf.data.Dataset.filter`
- For every sample
- Optionally, if you don't like TF libs, you can use `tf.py_func()`

[2] https://www.tensorflow.org/api_docs/python/tf/data/Dataset

Batching with `tf.data.Dataset`

Carnegie Mellon

- `tf.data.Dataset.shuffle(buffer_size)`
- `tf.data.Dataset.batch(batch_size)`
- `tf.data.Dataset.repeat(n_epochs)`

Not done yet, you need an `Iterator`

Carnegie Mellon

- Define an Iterator directly from `tf.data.Dataset` object
  - `iterator = tf.data.Dataset.make_one_shot_iterator()`
  - `iterator = tf.data.Dataset.make_initializable_iterator()`

Not done yet, you need an `Iterator`

- Define an Iterator directly from `tf.data.Dataset` object
  - `iterator = tf.data.Dataset.make_one_shot_iterator()`
  - `iterator = tf.data.Dataset.make_initializable_iterator()`
- Then we can get a batch of tensor easily:
  - `imgs, labels = iterator.get_next()`

Not done yet, you need an `Iterator`

Carnegie Mellon

- Define an Iterator directly from `tf.data.Dataset` object
  - `iterator = tf.data.Dataset.make_one_shot_iterator()`
  - `iterator = tf.data.Dataset.make_initializable_iterator()`
- Then we can get a batch of tensor easily:
  - `imgs, labels = iterator.get_next()`
- Can define each iterator for each dataset of train, val, test
  - Evalulation however we want during an epoch

Not done yet, you need an `Iterator`

Carnegie Mellon

- Define an Iterator directly from `tf.data.Dataset` object
  - `iterator = tf.data.Dataset.make_one_shot_iterator()`
  - `iterator = tf.data.Dataset.make_initializable_iterator()`
- Then we can get a batch of tensor easily:
  - `imgs, labels = iterator.get_next()`
- Can define each iterator for each dataset of train, val, test
  - Evalulation however we want during an epoch
- Or can define a shared iterator for train, val, test
  - `shared_iterator = tf.data.Iterator.from_structure()`

Not done yet, you need an `Iterator`

Carnegie Mellon

- Define an Iterator directly from `tf.data.Dataset` object
  - `iterator = tf.data.Dataset.make_one_shot_iterator()`
  - `iterator = tf.data.Dataset.make_initializable_iterator()`
- Then we can get a batch of tensor easily:
  - `imgs, labels = iterator.get_next()`
- Can define each iterator for each dataset of train, val, test
  - Evalulation however we want during an epoch
- Or can define a shared iterator for train, val, test
  - `shared_iterator = tf.data.Iterator.from_structure()`
- Do not forget to run the initializer for each iterator properly
  - Different in two cases above
  - Read the documentation and have fun coding for HW1

Outline

Grouping graph's ops into a dictionary

Carnegie Mellon

tf_graph_dict_trick.py

```
1  import tensorflow as tf
2
3  def build_dataset():
4
5  def build_tf_graph():
6    # create ops, variables, etc.
7    ops = {
8      "train": train_op,
9      "labels": labels,
10     "preds": get_predictions,
11   }
12   return ops
13
14 def main():
15   g = tf.Graph()
16   with g.as_default():
17     ops = build_tf_graph()
```

- Recall: Graph names are *different* from Python names
  - Can be lost when you build large graphs
  - You should use a Python `dict` to store these Python *handles*

Build a Model in a TF Graph

Carnegie Mellon

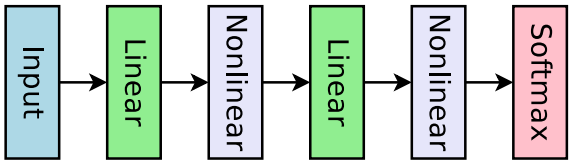
tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # x is a tensor of size [N, H, W, C]
3    N, H, W, C = tf.unstack(tf.shape(x))
4
5  def build_tf_graph():
6    images, labels = get_data_ops()
7    logits = feed_forward_net(images)
8    # LATER: loss function, train_op, etc.
```

- What are `images` and `labels`?
  - They are *TF ops*
- What happens in `N, H, W, C = tf.unstack(tf.shape(x))`?
  - `tf.shape(x)` is a *TF ops*.
  - `tf.unstack(...)` is a *TF ops*.
  - So are `N`, `H`, `W`, `C`
- Get used to them. Everything in TF is an ops.

Build a Model in a TF Graph

Carnegie Mellon



tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # x is a tensor of size [N, H, W, C]
3    N, H, W, C = tf.unstack(tf.shape(x))
4    x = tf.reshape(x, [N, H * W * C])  # flatten
5    for layer_id, next_dim in enumerate(dims):
6      curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
7      with tf.variable_scope("layer_{}".format(layer_id)):
8        w = tf.get_variable("w", [curr_dim, next_dim])  # w's name: "layer_2/w"
9      x = tf.matmul(x, w)
10     x = tf.nn.relu(x)
11   curr_dim = x.get_shape()[-1].value  # get_shape() returns a <list>
12   with tf.variable_scope("logits"):
13     w = tf.get_variable("w", [curr_dim, num_classes])  # w's name: "logits/w"
14     logits = tf.matmul(x, w)
15   return logits
```

- Flatten $\rightarrow$ (Linear $\rightarrow$ Nonlinear) $\times 3 \rightarrow$ logits

Loss Function

Carnegie Mellon

tf_feed_forward_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # The mess we just discussed
3
4  def build_tf_graph():
5    images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
6    logits = feed_forward_net(images)
7    # cross entropy loss function
8    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(
9      logits=logits, labels=labels)
10   loss = tf.reduce_mean(loss)
11   # LATER: train_op, etc.
```

- `tf.nn.(`*sparse*`_)softmax_cross_entropy_with_logits(logits, labels)` computes *all* losses.
- `tf.nn.sparse_softmax_cross_entropy_with_logits(...)` is *the correct way* to implement cross entropy loss with dense tensors
- `tf.nn.softmax_cross_entropy_with_logits(...)` is *the correct way* to implement cross entropy loss with one-hot tensors

Loss Function

Carnegie Mellon

tf_feed_forward_net.py

```
1  def build_tf_graph():
2    images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
3    logits = feed_forward_net(images)
4    # cross entropy loss function
5    probs = tf.nn.softmax(logits)
6    label_probs = tf.gather(probs, labels, axis=1)
7    loss = tf.log(label_probs)
8    # LATER: train_op, etc.
```

- Here is one (out of many) wrong ways!
  - Correct values, but much slower!
- Why?
  - $\log \frac{\exp\{\ell_i\}}{\sum_j \exp\{\ell_j\}}$ is *fused* into a kernel.

Train op

Carnegie Mellon

tf_feed_forward_net.py

```
1  def build_tf_graph():
2    images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
3    logits = feed_forward_net(images)  # a.k.a inference operation
4    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
5    # train_op
6    optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
7    train_op = optimizer.minimize(loss)
8    # LATER: predictions and accuracies
```

- `optimizer` is *not* the train op
- `optimizer.minimize(loss)` is the train op
  - Search for all TF variables
  - Traverse graph backward, assign a new *op* for gradient w.r.t each forward *op*
  - Variables are where the traverse stops (so are the backward *ops*)
  - Detail: new *op* created: `new_v = tf.assign_sub(v, lr *` ∇_v`)`,
  - a `tf.group` op is created to tie all those assign ops, and is returned.

Train op

Carnegie Mellon

tf_feed_forward_net.py

```
1  def build_tf_graph():
2    images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
3    logits = feed_forward_net(images)  # a.k.a inference operation
4    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
5    # train_op
6    optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
7    train_op = optimizer.minimize(loss)
8    # LATER: predictions and accuracies
```

- `optimizer` is *not* the train op
- `optimizer.minimize(loss)` is the train op
- If you call `sess.run([train_op])`, it will perform
  - Forward pass `images` and `labels`
  - Back(ward)-propagation
  - Update all variables using gradient descents.
- Due to TF Execution order, no op is run twice.

Predictions and Accuracies

Carnegie Mellon

tf_feed_forward_net.py

```
1  def build_tf_graph():
2    images, labels = get_data_ops()  # images, labels: [N, H, W, C], [N]
3    logits = feed_forward_net(images)
4    loss = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=logits, labels=labels)
5    optimizer = tf.train.GradientDescentOptimizer(learning_rate=1.0)
6    train_op = optimizer.minimize(loss)
7    # predictions and accuracies
8    preds = tf.argmax(logits, axis=1)
9    accus = tf.equal(preds, labels)  # TF boolean tensor
10   accus = tf.cast(accus, dtype=tf.int32)
11   accus = tf.reduce_sum(accus)
12   ops = {
13     "loss": loss,
14     "train_op": train_op,
15     "preds": preds,
16     "accus": accus,
17   }
18   return ops
```

- Everything is a TF ops
  - Get used to this concept / idea / paradigm
  - Get used to it anyhow.

What have we done?

Carnegie Mellon

tf_feed_forward_net.py

```
1  def build_tf_graph():
2    # The mess we discussed
3    ops = {
4      "loss": loss, "train_op": train_op, "preds": preds, "accus": accus
5    }
6    return ops
7
8  def main(_args):
9    g = tf.Graph()
10   with g.as_default():
11     ops = build_tf_graph()
12     with tf.Session() as sess:
13       sess.run(tf.global_variables_initializer()) # this is juts an ops!
14       for train_step in range(10000):
15         output = sess.run([ops["train_op"]])
```

- We built a TF graph with
  - Inference via a feed forward network, whose output are logits
  - Essential ops *to train* and *to use* the logits and other graph ops.

Can we build a ConvNet?

Carnegie Mellon

tf_conv_net.py

```
1  def feed_forward_net(x, dims=[256, 512, 128], num_classes=10):
2    # some mess you have seen
3
4  def conv_net(x, kernel_sizes=[3, 5, 7], num_channels=[128, 256, 512], num_classes=10):
5    # some mess you will fill in
6
7  def build_tf_graph():
8    images, labels = get_data_ops()
9    logits_feed_forward = feed_forward_net(images)
10   logits_conv = conv_net(images)
11   # some more mess
12   return ops
13
14 def main(_args):
15   with tf.Graph().as_default:
16     ops = build_tf_graph()
17     # even more mess
```

- No problem, only change the inference part:
  - Replace `feed_forward_net` with `conv_net`

Can we build a ConvNet?

Carnegie Mellon

tf_conv_net.py

```
1  def conv_net(x, kernel_sizes=[3, 5, 7], num_channels=[128, 256, 512], num_classes=10):
2    # x: tensor of size [N, H, W, C]
3    N, H, W, C = tf.unstack(tf.shape(x))
4    for layer_id, (k_size, next_c) in enumerate(zip(kernal_sizes, num_channels)):
5      curr_c = x.get_shape()[-1].value  # get_shape() returns a <list>
6      with tf.variable_scope("layer_{}".format(layer_id)):
7        # w's name: "layer_2/w"
8        w = tf.get_variable("w", [k_size, k_size, curr_c, next_c])
9      x = tf.nn.conv2d(x, w, padding="SAME")
10     x = tf.nn.relu(x)
11   x = tf.reshape(x, [N, -1])
12   curr_c = x.get_shape()[-1].value  # get_shape() returns a <list>
13   with tf.variable_scope("logits"):
14     w = tf.get_variable("w", [curr_c, num_classes])  # w's name: "logits/w"
15     logits = tf.matmul(x, w)
16   return logits
```

- No more flatten!
- Just change your `tf.matmul` into `tf.nn.conv2d()`
  - `padding="SAME"`

Other Graph Ops

Carnegie Mellon

tf_graph.py

```
def build_tf_graph():
  # ... omitted ...
  log_probs = tf.nn.sparse_softmax_cross_entropy_with_logits(
    logits=train_logits, labels=train_labels)
  train_loss = tf.reduce_mean(log_probs)
  l2_loss = tf.losses.get_regularization_loss()
  train_loss += l2_reg * l2_loss
  lr = tf.train.exponential_decay(init_lr, global_step * 64,
                                  50000, 0.98, staircase=True)
  optimizer = tf.train.MomentumOptimizer(
    learning_rate=lr, momentum=0.9)
  train_op = optimizer.minimize(train_loss, global_step=global_step)
  preds = tf.to_int32(tf.argmax(logits, axis=1))
  top5_preds = tf.nn.top_k(logits, k=5)
  ops = {
    "loss": train_loss, "train_op": train_op, "preds": preds, "top5_preds": top5_preds
  }
  return ops
```

- We have options to calculate accuracy on tensor-based or numpy-based later.

Outline

Training

Carnegie Mellon

tf_training.py

```
# data
images, labels = read_data(FLAGS.data_path)
# computational graph
g = tf.Graph()
with g.as_default():
ops = get_ops(images, labels)
# train
with tf.Session() as sess:
    sess.run(ops["train_iterator"])  # init dataset iterator
    for step in range(1, FLAGS.train_steps + 1):
        sess.run(ops["train_op"])
        if step % FLAGS.log_every == 0:
            get_eval_accuracy(ops, sess, step, "val") # validation
    # done with training, now testing
    get_eval_accuracy(ops, sess, step, "test")
```

- Remember: as of yet, you must always have a `tf.Session` object to train

Next

- Dynamic Graph
- Save and Restore
- Using pre-trained weights
- `tf.Estimator` framework (optional)
- Tensorboard
- Your desired topics