Решения задачи машинного обучения состоит из нескольких основных этапов:

1. Сбор данных
2. Подготовка структуры данных
3. Разработка модели
4. Обучение
5. Тестирование
6. *подготовка к production

Одним из важных процессов является работа с входными данными, который состоит из следующих этапов:

1. **Extract** / Извлечение
2. **Transform** / Трансформация
3. **Load** / Загрузка

В Tensorflow для этого есть несколько библиотек, которые позволяют не только выполнить вышеописанные этапы, но и оптимизировать данный процесс.

TF Queue

Tensorflow Queue - это структура данных для обработки коллекций (известные всем Priority Queue, FIFO Queue)
В TF queue используется для загрузки данных, их обработки и формирования батчей.
Есть несколько вариантов Queue:

- FIFOQueue
- PriorityQueue
- PaddingFIFOQueue
- RandomShuffleQueue

Priority Queue:
PQ в TF представляет собой такую же структуру данных, что есть в классическом понимании: у каждого элемента в очереди есть свой приоритет - чем ниже значение, тем выше приоритет. *(Для triplet можно объединить по priority, чтобы вместе доставались)*
Пример использования:

Конструктор

```
__init__(
    capacity,
    types,
    shapes=None,
    names=None,
    shared_name=None,
    name='priority_queue'
)
```

```
priority = tf.placeholder(shape=(None), dtype=tf.int64)
image_path = tf.placeholder(shape=(None), dtype=tf.string)
image_label = tf.placeholder(shape=(None), dtype=tf.int64)

queue = tf.PriorityQueue(capacity=10, types=[tf.string, tf.int64], shapes=[(), ()])

enqueue_op = queue.enqueue_many([priority, image_path, image_label])
dequeue_op = queue.dequeue()

session = tf.Session()

session.run(enqueue_op, feed_dict={priority: [2, 1, 4, 3],
                                   image_path: ['item2', 'item1', 'item4', 'item3'],
                                   image_label: [0, 1, 1, 0]
                                   })

output = session.run(dequeue_op)
# [1, 'item1', 1]
```

PaddingFIFOQueue:
FIFO очередь, отличие которой заключается в том, что тензоры в батче могут иметь разный размер, но при получении элементов из очереди (dequeue) тензоры будут заполняться нулями до максимальной длины.
Пример использования:
Конструктор

```
__init__(
    capacity,
    dtypes,
    shapes,
    names=None,
    shared_name=None,
    name='padding_fifo_queue'
)
```

```
sequence = tf.placeholder(shape=(None), dtype=tf.int64)
sequence_label = tf.placeholder(shape=(), dtype=tf.int64)
queue = tf.PaddingFIFOQueue(capacity=20, dtypes=[tf.int64, tf.int64], shapes=[(None,), ()])
enqueue = queue.enqueue([sequence, sequence_label])
dequeue = queue.dequeue_many(n=2)

session = tf.Session()

session.run(enqueue, feed_dict={sequence: [1, 2, 3],
                                sequence_label: 0})
session.run(enqueue, feed_dict={sequence: [1, 2],
                                sequence_label: 1})
output = session.run(dequeue)

print(output)
# [array([[1, 2, 3], [1, 2, 0]]),
# array([0, 1])]
```

RandomShuffleQueue:
Очередь, где элементы достаются в случайном порядке. Размер элемента, который попадает в очередь должен быть фиксированным. Но при создании данной очереди необходимо указать минимальное количество элементов, которое должно остаться после операции извлечении элементов из очереди (dequeue) - `min_after_dequeue`. Чтобы выполнение кода не зависало, необходимо дополнять очередь элементами, чтобы остаток был всегда >= `min_after_dequeue`.
Пример использования:
Конструктор

```
__init__(
    capacity,
    min_after_dequeue,
    dtypes,
    shapes=None,
    names=None,
    seed=None,
    shared_name=None,
    name='random_shuffle_queue'
)
```

```
image_path = tf.placeholder(shape=(None), dtype=tf.string)
image_label = tf.placeholder(shape=(None), dtype=tf.int64)

queue = tf.RandomShuffleQueue(capacity=10, dtypes=[tf.string, tf.int64],
                              shapes=[(), ()], min_after_dequeue=0)
enqueue_op = queue.enqueue_many([image_path, image_label])
dequeue_op = queue.dequeue()

session = tf.Session()

session.run(enqueue_op, feed_dict={image_path: ['item2', 'item1', 'item4', 'item3'],
                                   image_label: [0, 1, 1, 0]
                                   })

output = session.run(dequeue_op)
# random item will be chosen

print(output)
```

FIFOQueue:
FIFO очередь, работающая по принципу “first in first out”. Наиболее часто используется для процесса подготовки данных во время обучения.
Пример использования:
Конструктор

```
__init__(
    capacity,
    dtypes,
    shapes=None,
    names=None,
    shared_name=None,
    name='fifo_queue'
)
```

```
image_path = tf.placeholder(shape=(None), dtype=tf.string)
image_label = tf.placeholder(shape=(None), dtype=tf.int64)

queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string, tf.int64],
                     shapes=[(), ()])
enqueue_op = queue.enqueue_many([image_path, image_label])
dequeue_op = queue.dequeue()

session = tf.Session()

session.run(enqueue_op, feed_dict={image_path: ['item2', 'item1', 'item4', 'item3'],
                                   image_label: [0, 1, 1, 0]
                                  })

output = session.run(dequeue_op)
# ['item2', 0]
```

==========

Основное преимущество использования очередей в TF - это то, что процесс обработки входных данных можно распараллелить.
Если говорить о данных, которые являются входными для обучения, то их количество может быть очень большим, тогда для увеличения скорости их обработки для подачи на вход модели имеет смысл распараллеливать процесс пополнения очереди. Это делается с помощью **tf.train.Coordinator** и **tf.train.QueueRunner.**

tf.train.Coordinator

Coordinator отвечает за управление потоками, а именно за прекращение выполнения задач действующими потоками одновременно. Помимо этого Coordinator контролирует появление ошибок, если один из потоков выдают исключение, то все остальные потоки должны остановить свою работу, либо определить набор действий необходимые сделать при появлении ошибки.
Coordinator применяется не только для работы с очередями, но и для выполнения python кода, который можно распараллелить на несколько потоков.

tf.train.QueueRunner

QueueRunner используется для создания нескольких потоков, в которых будет происходить пополнение очереди.
QueueRunner и Coordinator работают в связке.

Пример использования без батча:

```
NROF_THREADS = 4
NROF_ITERATIONS = 4

image_path = ['item2', 'item1', 'item4', 'item3']
image_label = [0, 1, 1, 0]

queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string, tf.int32], shapes=[(), ()])
enqueue_op = queue.enqueue_many([image_path, image_label])
dequeue_op = queue.dequeue()

session = tf.Session()

# QueueRunner for running enqueue op in parallel using NROF_THREADS
queue_runner = tf.train.QueueRunner(queue, [enqueue_op] * NROF_THREADS)
tf.train.add_queue_runner(queue_runner)

# Coordinator for launching QueueRunner
coordinator = tf.train.Coordinator()

enqueue_threads = queue_runner.create_threads(session, coord=coordinator, start=True)

for step in range(NROF_ITERATIONS):
    if coordinator.should_stop():
        break
    image_path_batch, label_batch = session.run(dequeue_op)

coordinator.request_stop()
coordinator.join(enqueue_threads)
```

Пример использования для батча:

```
NROF_THREADS = 4
NROF_ITERATIONS = 2
BATCH_SIZE = 2

image_path = tf.placeholder(shape=(None), dtype=tf.string)
image_label = tf.placeholder(shape=(None), dtype=tf.int64)

queue = tf.FIFOQueue(capacity=10, dtypes=[tf.string, tf.int64], shapes=[(), ()])

enqueue_op = queue.enqueue_many([image_path, image_label])

paths_images_and_labels = []
for _ in range(NROF_THREADS):
    filename, label = queue.dequeue()
    paths_images_and_labels.append([filename, label])

# Fill Queue to create a batch of examples
path_batch, label_batch = tf.train.batch_join(paths_images_and_labels,
                                              batch_size=BATCH_SIZE,
                                              enqueue_many=False)

session = tf.Session()

# Coordinator for launching QueueRunner
coordinator = tf.train.Coordinator()
# Run all queues that were launched by threads
enqueue_threads = tf.train.start_queue_runners(coord=coordinator, sess=session)

session.run(enqueue_op, feed_dict={image_path: ['item2', 'item1', 'item4', 'item3'],
                                   image_label: [0, 1, 1, 0]
                                   })

for step in range(NROF_ITERATIONS):
    if coordinator.should_stop():
        break
    image_path_output, label_output = session.run([path_batch, label_batch])
    print(image_path_output, label_output)

coordinator.request_stop()
coordinator.join(enqueue_threads)
```

Выше представленные примеры не содержат в себе сложную логику обработки входных данных, которые после необходимо положить в очередь, но на практике чаще всего входные данные требуют предобработку, прежде чем модель начнет с ними работать. И при таком раскладе распараллеливание помогает повысить производительность.

Note: Параметр *enqueue_many* в методе **tf.train.batch_join** если помечен как *True*, то paths_images_and_labels считается одним элементом из датасета, а значение *False*, как раз отвечает за то, что это батч, а не единственный элемент.

Python function injection:

Для внедрения python код в Tensorflow, как в статичный, так и в динамичный граф (1.+/ 2.0 версии) существует метод ***tf.py_function***

```python
def read_and_augmment_image(filename, label):
    def read_cv2(path_):
        image = cv2.imread(str(np.core.defchararray.decode(path_.numpy())))
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        return image

    img_raw = tf.py_function(read_cv2, [filename], np.uint8)
    img_tensor = tf.image.decode_image(img_raw, channels=3)
    img_tensor.set_shape((None, None, 3))
    img_final = tf.image.resize(img_tensor, [256, 256])
    img_final = img_final / 255.0

    return img_final, label
```

tf.py_function принимает на вход:

1) функцию на python code
2) набор параметров, которые являются аргумента в python функции
3) типы данных, которые получаются на выходе python функции.

Такой подход может быть полезен, когда необходимо внедрить какие-то свои операции, которых нет в Tensorflow. Это простой способ решения подобных вопросов, для более продвинутого подхода можно сделать с помощью реализации операций на C++, и уже его внедрения в Tensorflow код.