TensorFlow 教程：从基础到 CNN 和 seq2seq

阮 翀

2018/02/28

（本讲义中的代码均已开源：https://github.com/soloice/tf-tutorial）

提纲

- 预备知识
  - TF 相关背景
- 基础知识
  - 定义常量和变量
  - Graph 和 Session
  - 占位符
- CNN
  - MNIST 手写数字识别
- RNN
  - 普通 RNN：字符语言模型
  - seq2seq 实例：给定一个数字序列，删除其中的奇数

预备知识

- 安装
  - 参考 https://www.tensorflow.org/install/
  - GPU 版安装较繁琐，需要先装 CUDA & cuDNN
  - CPU 版可以直接 pip install tensorflow
  - Mac/Linux
    - 命令行里直接安装即可，Python 2/3 均可
  - Windows
    - 只支持 Python 3.5+
    - 推荐安装 Anaconda（一个集成了 scipy/numpy 等科学计算包的 Python 发行版），各种依赖一键解决

预备知识

- 文档
  - 官方文档一般都很详细，遇到问题多查文档
  - https://www.tensorflow.org/
    - 右上角有搜索框，搜关键字即可
    - 极客学院有中文翻译版的文档，不过有能力的话最好还是看英文原文
      - http://wiki.jikexueyuan.com/project/tensorflow-zh/
  - 当然还有 StackOverflow

预备知识

- 版本
  - 现在最新的版本大约是 r1.5
  - 由于深度学习发展很快，各个版本的差异可能较大
    - 例如 API 名称、函数参数的顺序等都可能有变动
    - 1.0 版本和之前的版本不兼容
      - 网上很多代码都是基于 tf r0.12 的 API，不要参考
    - 1.0 版之后 TF 发布了一个保持接口兼容性的承诺
      - 大部分 API 都不会再变了，保持后向兼容
      - 但是不包括 tf.contrib 里的部分
        - 这里的内容是其他开发者贡献的，而非 TF 官方
        - 其中效果好、被广泛使用的部分会逐步被 TF 官方合并
      - 详见 https://www.tensorflow.org/programmers_guide/version_compat

Aliases:

- Class `tf.contrib.rnn.BasicLSTMCell`
- Class `tf.nn.rnn_cell.BasicLSTMCell`

预备知识

- bug
  - 有时候程序运行结果不符合预期不是你的错，而是框架的 bug
    - 毕竟框架只有几年，不像 C++ 的用了几十年的库一样可靠
  - bug 举例：
    - GPU 版和 CPU 版 global_step（模型迭代的 batch 数）更新方式不同
    - Beam Search 有时第一次遇到结束符不停止，连续输出两个结束符才停
    - 不确定现在是否已经修复，用的时候多长点儿心眼
  - 发现 bug 了可以在 GitHub 上给官方提 issue
    - 但是要有耐心：他们可能没看懂就把你的 issue 关掉，因为小白太多了，经常有人提的问题是自己程序的 bug 而不是 TF 的 bug
  - 其实各大深度学习框架 bug 都很多，TF 自从 1.2 版本之后已经算是 bug 很少的了，习惯就好

预备知识

- 学习资料
  - 官网教程
    - https://www.tensorflow.org/tutorials/
    - 说实话谷歌做这个东西没用心
      - 有些部分跨度比较大
      - 更新不及时，有些 API 已经逐渐弃用了教程里还没改
  - 谷歌新上线的机器学习教程
    - 看了一下目录，学习曲线比较平缓
    - 偏重于传统机器学习方法，以及 TF 的使用
    - 神经网络部分只有 MLP，不涉及深度学习
    - https://developers.google.cn/machine-learning/crash-course/

预备知识

- 学习资料
  - CS 20
    - 斯坦福的 TF 课程，很详细，强烈推荐
    - https://web.stanford.edu/class/cs20si/
  - Udacity 上的深度学习
    - 这个其实也是谷歌出的，比官网上的教程好很多
    - https://cn.udacity.com/course/deep-learning--ud730
  - 莫烦的视频
    - 周沫凡，一个做智能交通的 PhD
    - https://morvanzhou.github.io/tutorials/machine-learning/tensorflow/

基础知识

- 计算图的定义和执行相分离
  - 先在计算图中定义计算流程（在 TF 中叫做 Operation）
  - 然后在数据上执行具体计算
  - *新出的 eager 模式改进了这一点，暂不讨论
- 最基本的例子

```
import tensorflow as tf

# Construct the computation graph
a = tf.constant(3.0)
b = tf.constant(2.0)
c = a + b

# Then Execute it
sess = tf.Session()
c_value = sess.run(c)
print(c_value)
```

运行结果： 5.0

基础知识

- 数据类型 DType
  - 可选 tf.int32, tf.float32 等
  - 和 numpy 兼容
    - 其实就是 np.int32, np.float32 等
    - 以下程序输出 True

```
import tensorflow as tf
import numpy as np
print(tf.int32 == np.int32)
```

  - 更多类别详见 https://www.tensorflow.org/versions/master/api_docs/python/tf/DType

基础知识

```
tf.constant(
    value,
    dtype=None,
    shape=None,
    name='Const',
    verify_shape=False
)
```

- 定义常量
  - value 就是常量的值，必须给定
  - 其他参数可选
  - 可以给常量起名字
    - 示例

```
a = tf.constant(3.0)
b = tf.constant(2.0, name="variable_b")
print("a.name = ", a.name)
print("b.name = ", b.name)
```

```
a.name =  Const:0
b.name =  variable_b:0
```

  - 名称格式为 <字符串>:<数字>，因为有的 Op 有多个返回值
    - 举例：若 x, y = SomeOperation(..., name="my_name")
    - 则 x 和 y 的名字分别叫 my_name:0 和 my_name:1
- https://www.tensorflow.org/versions/master/api_docs/python/tf/constant

基础知识

- 定义变量
  - 两种方式
    - tf.get_variable() 或 tf.Variable()
    - 注意 tf.Variable() 中 V 是大写的
      - 实际上这是一个内部很复杂的类
  - 内容很多，详见
    - https://www.tensorflow.org/versions/master/api_docs/python/tf/get_variable
    - https://www.tensorflow.org/versions/master/api_docs/python/tf/Variable

基础知识

- 定义变量
  - tf.Variable() 需要给定初值
  - tf.get_variable 要给定 initializer
    - 指明如何初始化这个变量
    - 可以是一个给定的值
    - 也可以从一个分布中去采样
    - 如果未指明，默认使用 glorot_uniform_initializer
      - https://stackoverflow.com/questions/37350131/what-is-the-default-variable-initializer-in-tensorflow

```
x = tf.Variable([0.1, 0.2], name='x')
y = tf.get_variable(name='y', shape=[2],
                    initializer=tf.constant_initializer([1.0, 2.0]))
z = tf.get_variable(name='z', shape=[2, 3],
                    initializer=tf.random_normal_initializer())

sess = tf.Session()
sess.run(tf.global_variables_initializer())
x_value = sess.run(x)
y_value = sess.run(y)
z_value = sess.run(z)
print("x = ", x_value)
print("y = ", y_value)
print("z = ", z_value)
```

```
x =  [ 0.1  0.2]
y =  [ 1.  2.]
z =  [[ 0.11233806  0.05913079  0.99890053]
 [ 1.58906424  0.41413647 -1.57551074]]
```

基础知识

- Graph 和 Session
  - Graph
    - 用来放置各种 Op，在定义计算图的时候使用
    - 计算图上的节点都是符号，只说明节点之间如何依赖
      - 例如 c = a + b 的意思是，假如要计算 c，先要取出 a 和 b 的值，然后相加得到 c
      - 即便 a 和 b 的值已知，这时也不会算出 c 的值
  - Session
    - 用来执行具体的计算，取出计算图中某个节点的值
    - 这里才会分配内存/显存存储变量的值，之前的变量都是符号

基础知识

- Graph
  - 每定义一个变量/常量就是往一个 Graph 对象上添加一个节点
  - 如果不显式声明 Graph，就会使用默认的 Graph
  - 以下两种定义计算图的方式是等价的

```
c = tf.constant(5.0)
```

```
with tf.get_default_graph().as_default():
    c = tf.constant(5.0)
```

  - 理论上可以定义多个 Graph，每个 Graph 上放一些 Op
  - 但是实际上最好不要操作多个 Graph
    - TF 中多 Graph 的用法是个谜
  - 如果需要有多个不同的图怎么办？
    - 都放在一个 Graph 上，几个模块之间不连通即可

基础知识

- Session
  - 用于执行具体的计算
  - 初始化
    - 只有变量要初始化，常量不需要

```
init_op = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init_op)
```

    - 与之对应地，还有 tf.local_variables_initializer()
      - 但是一般用不到，正常方式定义的变量都是 global variables

基础知识

- Session
  - 用于执行具体的计算
  - 初始化
    - 网上有的教程写的是 tf.initialize_all_variables()，针对的是较老的 TF 版本，现在已经废弃了
    - 对于变量 x，也可以用 sess.run(x.initializer) 来初始化
  - 如果用 tf.Session() 打开了一个 session，之后就不能再修改计算图了，只能执行图中已定义的 Op
  - 在 shell 里调试程序时，可以用 tf.InteractiveSession() 打开一个 session，这时还可以再向计算图中添加新的节点
  - 另：TF 默认占据一台机器上的全部计算资源，可以设置一些 config 来减少资源占用，详见 http://blog.csdn.net/u012436149/article/details/53837651

基础知识

- 变量重用
  - tf.Variable() 会自动对同名变量重命名，定义两次会得到两个变量

```
y = tf.Variable(0, name="y", dtype=tf.float32)
another_y = tf.Variable(0, name="y", dtype=tf.float32)
```

```
y:0
y_1:0
```

  - tf.get_variable() 则会先检测同名变量是否存在，如果存在则会报错

```
x = tf.get_variable("x", shape=[2, 3], dtype=tf.float32)
another_x = tf.get_variable("x", shape=[2, 3], dtype=tf.float32)
```

    - 报错内容为：ValueError: Variable x already exists, disallowed. Did you mean to set reuse=True in VarScope? Originally defined at:

基础知识

```
with tf.variable_scope("foo", reuse=None):
    x1 = tf.get_variable("x", shape=[2, 3])

with tf.variable_scope("foo", reuse=True):
    x2 = tf.get_variable("x", shape=[2, 3])

assert x1 == x2
```

- 变量重用
  - 正确方式
    - variable scope + reuse
  - 第一次定义某个变量时，把 reuse 设置成 None
    - 写成 False 也可以，但是 TF 的习惯是写 None
    - 也可以不设置 reuse 选项，此时默认 reuse=None
  - 第二次定义该变量时，把 reuse 置为 True
    - 此时不会新定义变量，而是会把原先定义的变量再取出来
  - 此时 x1 和 x2 是同一个变量，在 Session 中执行的时候只分配一次内存
  - 在最新版本的 TF 中，只需要设置 reuse=tf.AUTO_REUSE 即可
    - 如果变量未定义则定义新变量，否则自动重用，非常方便
  - https://www.tensorflow.org/versions/master/programmers_guide/variables

基础知识

- TF 中的域机制
  - 两种
    - 变量域（variable scope）和名称域（name scope）
  - 为什么需要域？
    - 便于组织和管理变量
  - 考虑定义一个多层 MLP
    - 每一层都有一个权重矩阵 W 和偏置项 b
    - 如果把变量命名成 W1, b1, W2, b2, ..., Wn, bn 有点蠢
      - 所有变量都是平级的，体现不出层次关系
    - 如果加上域机制
      - 命名为 layer1/W, layer1/b, layer2/W, layer2/b, ...，层次关系更清楚
      - 可以每个域内使用同一个学习率/初始化方法等，更加模块化

基础知识

- TF 中的域机制
  - 对于更复杂的结构（如 ResNet, LSTM 等）效果更好
    - scope 可以多层嵌套
    - 例如 seq2seq 模型中，可以按照如下方式命名：
      - decoder/layer1/forget_gate/bias
  - name scope 和 variable scope 的联系和区别
    - 共同点：两种 scope 都用于把变量组织成模块，在 TensorBoard 中可视化的时候都可以把一个模块缩成一个节点来显示，使得模型结构看起来更清楚
    - 不同点：tf.get_variable 会忽略 name scope，只看 variable scope
  - 其他细节
    - http://blog.csdn.net/qq_19918373/article/details/69499091
  - 多用 tf.get_variable()，少用 tf.Variable()
    - 后者自动重命名重名变量，对于复杂模型有安全隐患

基础知识

- TF 中的域机制
  - 举例

```
with tf.name_scope("my_scope"):
    v1 = tf.get_variable("var1", [1], dtype=tf.float32)
    v2 = tf.Variable(1, name="var2", dtype=tf.float32)
    a = tf.add(v1, v2)

print(v1.name)  # var1:0
print(v2.name)  # my_scope/var2:0
print(a.name)   # my_scope/Add:0
```

```
with tf.variable_scope("my_scope"):
    v1 = tf.get_variable("var1", [1], dtype=tf.float32)
    v2 = tf.Variable(1, name="var2", dtype=tf.float32)
    a = tf.add(v1, v2)

print(v1.name)  # my_scope/var1:0
print(v2.name)  # my_scope/var2:0
print(a.name)   # my_scope/Add:0
```

基础知识

- 变量赋值
  - 机器学习中通常不需要手动设置变量的值，而是通过梯度下降等方式来更新变量
  - 如果确实需要赋值（例如从命令行指定学习率），可以使用 tf.assign

```
update_op = tf.assign(x, 5)
sess.run(update_op)
```

基础知识

- 占位符（placeholder）
  - 有时候一些值是未知的，但是模型需要依赖这些值来进行后续计算
  - 例如线性回归
    - 我们定义的变量是直线的斜率和截距
    - 但是数据点的坐标在定义模型时是未知的
  - 这时可以使用占位符来代表数据点的坐标
  - 当得到数据值以后（例如从文件中读取到），可以用一个字典把数据的真实值传进去

基础知识

- 占位符

```
x = tf.placeholder(tf.float32, shape=(1024, 1024))
y = tf.matmul(x, x)

with tf.Session() as sess:
  print(sess.run(y))  # ERROR: will fail because x was not fed.

  rand_array = np.random.rand(1024, 1024)
  print(sess.run(y, feed_dict={x: rand_array}))  # Will succeed.
```

- https://www.tensorflow.org/versions/master/api_docs/python/tf/placeholder

基础知识

- 占位符
  - 其实别的 Tensor 也可以强行 feed
    - 占位符本来就是一种必须要 feed 的特殊的 Tensor
  - 以下程序里，把 1.0 强行 feed 给 a 之后，c 就变成了 1.0 + 3.0 = 4.0
  - 注意 feed 过后 a 还是 2.0
    - 只是计算 c 时用 1.0 代替 a 本来的值，并不会向 a 赋值

```
a = tf.constant(2.0)
b = tf.constant(3.0)
c = a + b

with tf.Session() as sess:
    print(sess.run(c, feed_dict={a:1.0}))
```

基础知识

- 优化器（Optimizer）
  - 定义好模型之后，就可以用梯度下降来更新模型参数了
  - 以最简单的梯度下降为例
    - 先定义 Optimizer
      - 说明如何更新模型参数，这里是梯度下降
    - 然后 optimizer.minimize 定义了一个 Op
      - 对什么目标函数做梯度下降
      - 对哪些变量做更新（用参数 var_list 控制，默认对全部 Trainable 的参数都做）

```
optimizer = tf.train.GradientDescentOptimizer(0.5) # 梯度下降法，学习速率为0.5
train_op = optimizer.minimize(loss) # 训练目标：最小化损失函数
```

  - 详见 https://www.tensorflow.org/versions/master/api_docs/python/tf/train/Optimizer

基础知识

- 一个完整的例子：线性回归
- 定义计算图

```python
import tensorflow as tf
import numpy as np


with tf.get_default_graph().as_default():
    x = tf.placeholder(tf.float32, [], name="x")
    y = tf.placeholder(tf.float32, [], name="y")
    k = tf.get_variable("k", shape=[])
    b = tf.get_variable("b", shape=[])
    loss = tf.square(k * x + b - y, name="square_error")

    optimizer = tf.train.GradientDescentOptimizer(0.5)
    train_op = optimizer.minimize(loss)
```

基础知识

- 一个完整的例子：线性回归
- 在直线 y=3x-1 附近生成随机数据点并做随机梯度下降

```
with tf.Session() as sess:
    # Initialize `k` and `b`
    sess.run(tf.global_variables_initializer())
    for i in range(100):
        # y = 3x-1 + noise
        data_x = np.random.random()
        data_y = 3.0 * data_x - 1.0 + 0.0001*np.random.randn()

        k_val, b_val, loss_val, _ = sess.run([k, b, loss, train_op],
                                             feed_dict={x: data_x, y: data_y})
        if i % 20 == 0:
            print("k, b, loss: ", k_val, b_val, loss_val)
```

基础知识

- 一个完整的例子：线性回归
- 在直线 y=3x-1 附近生成随机数据点并做随机梯度下降

```
k, b, loss:  0.222541 0.915134 1.14494
k, b, loss:  2.53555 -0.927716 0.00320017
k, b, loss:  2.80536 -0.896746 0.00643553
k, b, loss:  2.92283 -0.985189 8.28559e-05
k, b, loss:  2.99505 -0.993757 3.91142e-05
```

基础知识

- 其他模块
  - Tensorboard
    - 用于对 loss/参数分布/词向量等可视化，便于了解训练过程
  - 检查点（checkpoint）
    - 保存模型到硬盘上，之后可以再次从检查点恢复模型参数
    - 大规模训练时可以用一个进程做训练，定期保存检查点，另一个进程做验证
  - Dataset API
    - 给模型喂数据的另一种方法
      - 与 placeholder + feed dict 平行，互不干扰
    - 模型的输入是一个 Dataset（一般可以从文件流开始构造），TF 会自动读取数据并作适当转换（例如 shuffle/加噪声/归一化/左右翻转等）形成 batch 供模型使用

基础知识

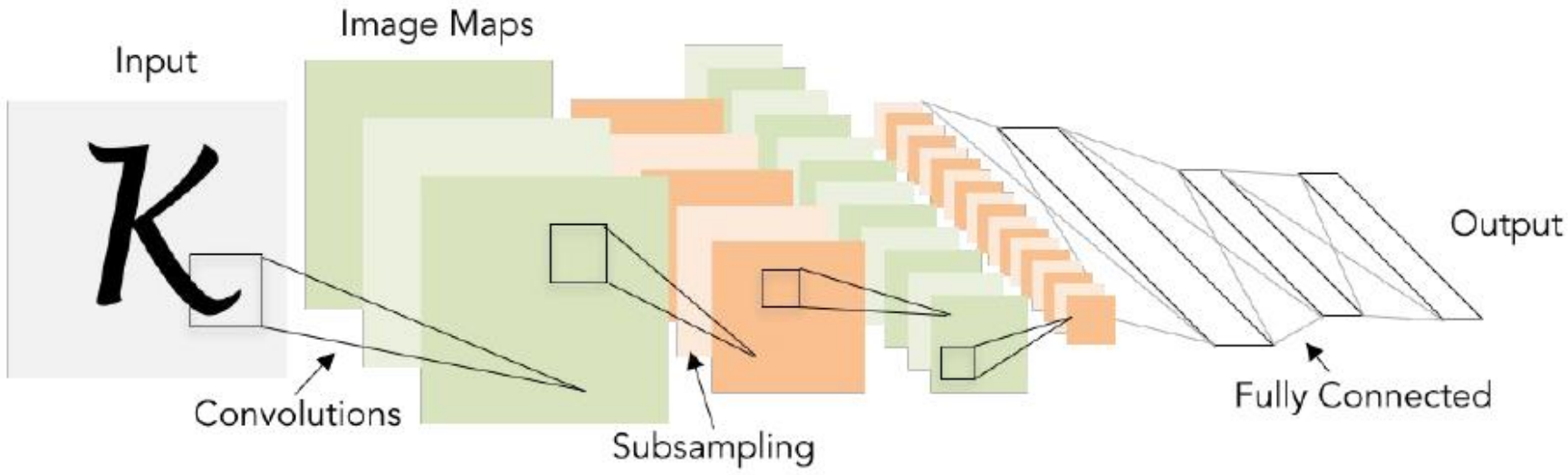
- 其他模块
  - Queue
    - 有时数据规模较大，不能全部放进内存
    - 将读取数据和模型计算分拆到两个进程中，防止计算和 I/O 互相阻塞降低效率
    - http://wiki.jikexueyuan.com/project/tensorflow-zh/how_tos/threading_and_queues.html

MNIST

- 手写数字数据集
- http://yann.lecun.com/exdb/mnist/
- 所有数字的图像都是 28*28 的灰度图
- 训练集：6w（也有划分成 5.5w 训练集 + 5k 验证集的）
- 测试集：1w

train-images-idx3-ubyte.gz:	training set images (9912422 bytes)
train-labels-idx1-ubyte.gz:	training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz:	test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz:	test set labels (4542 bytes)

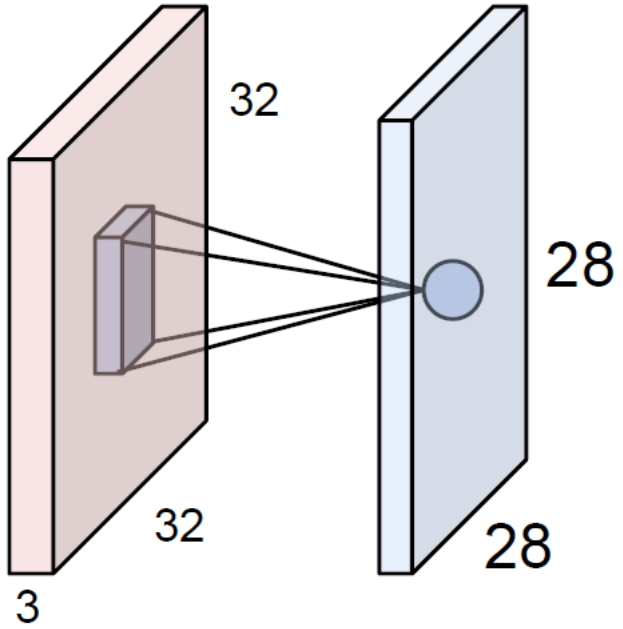
LeNet



LeNet-5

http://cs231n.stanford.edu/syllabus.html

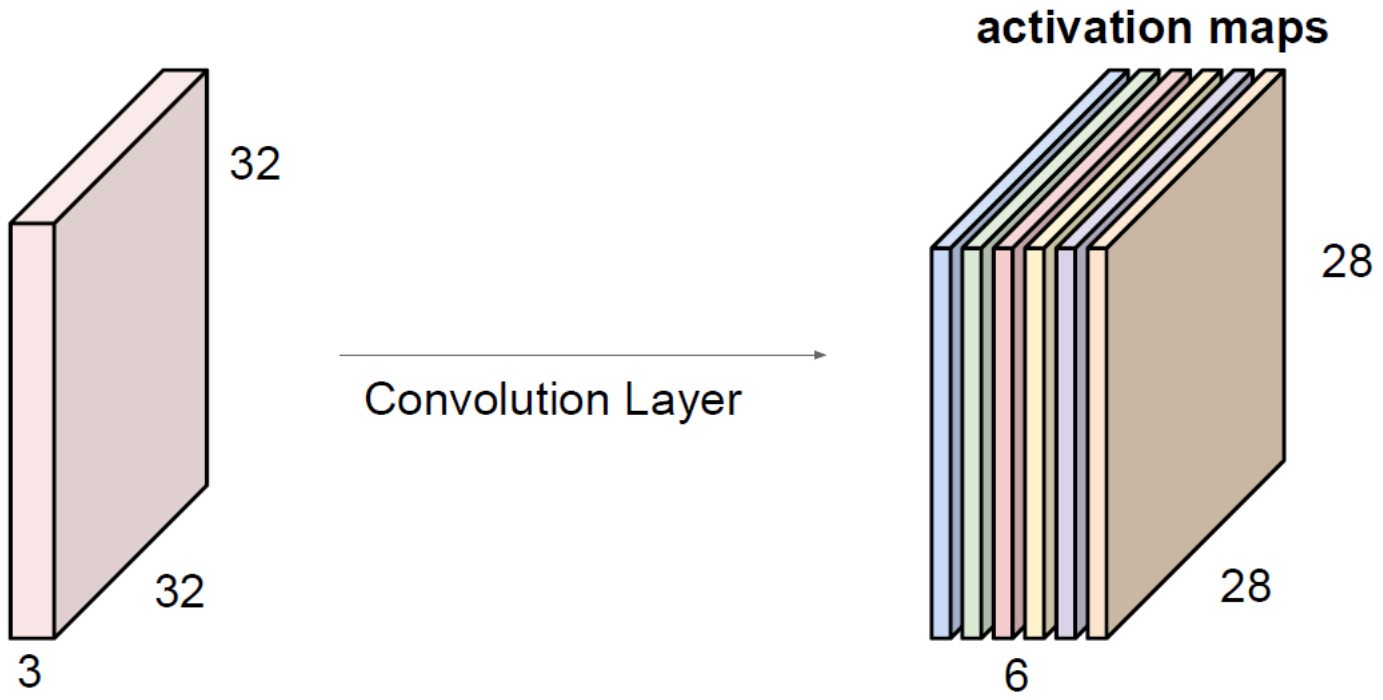
CNN



- 卷积用于提取空间平移不变性
  - 同样的卷积核在整张图上滑动
- 单个卷积核（也叫 filter）
  - 假如图像大小是 32*32*3
  - 卷积核大小是 5*5*3
  - 输出图像的大小是 (32-5+1)*(32-5+1)=28*28

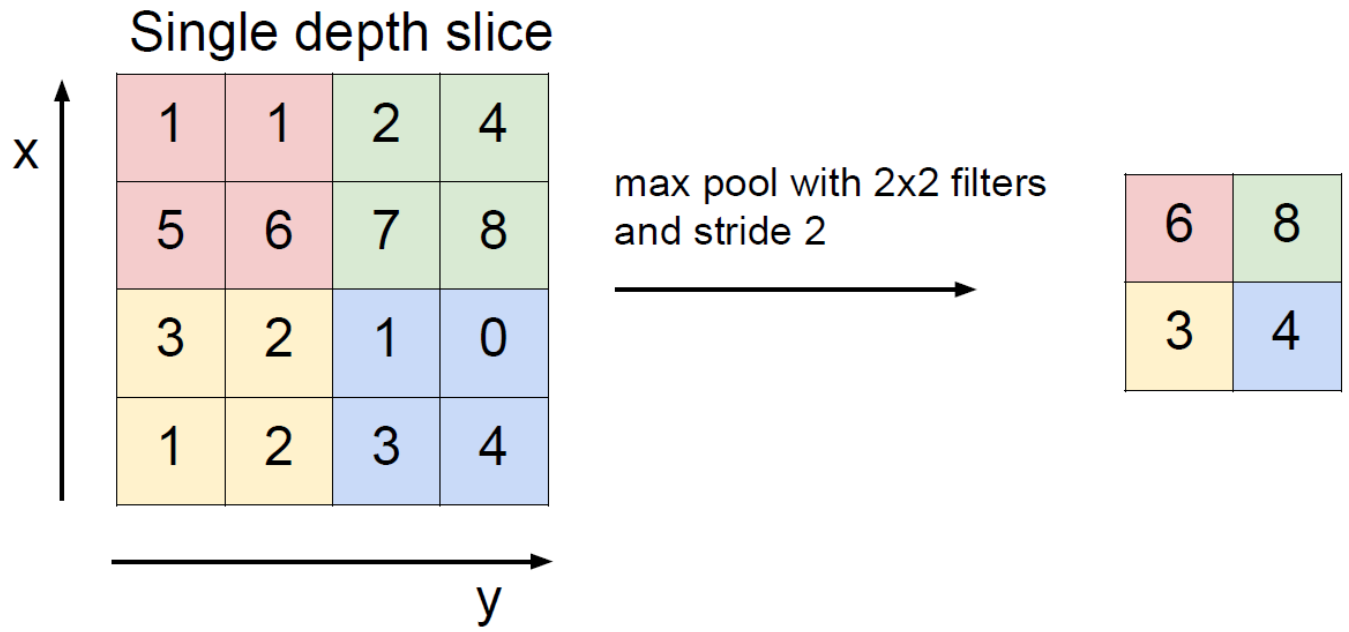
CNN

- 多个卷积核堆叠
  - 此处有 6 个，输出图像的形状是 28*28*6



CNN

- （最大）池化用于提取尺度和旋转不变形
  - 轻微的角度和尺度差异在池化后会消失



CNN

- 典型 CNN 结构

[(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX
where N is usually up to ~5, M is large, 0 <= K <= 2.

CNN

- TF 代码实现
- 可以写一些辅助函数，修改默认的初始化方法

```python
import tensorflow as tf
import time
from tensorflow.examples.tutorials.mnist import input_data


def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)


def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

CNN

- TF 代码实现
- 可以把 TF 里的卷积和池化操作封装起来，更好用
  - 卷积是 tf.nn.conv2d
  - 我们只使用 5*5 的卷积核，图像宽和高都相等，滑动步长取 1

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')


def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
```

CNN

- TF 代码实现
- 类似地，可以封装一下全连接层（使用 scope）
  - 只需指明输出通道数即可，输入通道数可以从参数的形状推断出来
  - 参数形状为 [B, H, W, C]
    - 四维张量，四个轴分别代表批大小、图像的高度、宽度、通道数

```
def fc_layer(x, out_dim=10, scope="fc"):
    with tf.variable_scope(scope):
        in_dim = int(x.get_shape()[-1])
        W_fc = weight_variable([in_dim, out_dim])
        b_fc = bias_variable([out_dim])
        fc = tf.nn.relu(tf.matmul(x, W_fc) + b_fc)
        return fc
```

CNN

- TF 代码实现
- 同理，还有卷积层
  - 每做一次卷积（也可以多做几次），就跟着一次池化

```python
def conv_and_pool(x, kernel_size=5, out_channel=32, scope="conv_layer"):
    with tf.variable_scope(scope):
        in_channel = int(x.get_shape()[-1])
        # print(type(in_channel), in_channel)
        # print(x.get_shape())
        W_conv = weight_variable([kernel_size, kernel_size, in_channel, out_channel])
        b_conv = bias_variable([out_channel])

        h_conv = tf.nn.relu(conv2d(x, W_conv) + b_conv)
        h_pool = max_pool_2x2(h_conv)
        return h_pool
```

CNN

- TF 代码实现
- 读数据可以用 tensorflow.examples.tutorials.mnist 中的接口

```
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
print(mnist.train.images.shape)
print(mnist.test.images.shape)
```

- 输出如下

```
(55000, 784)
(10000, 784)
```

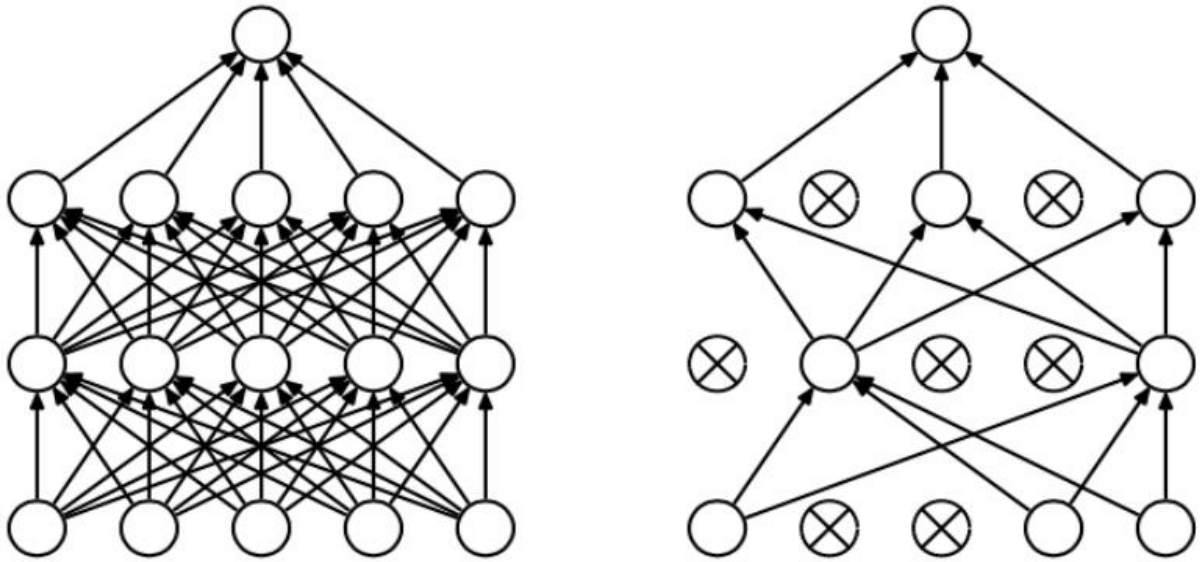
CNN

- TF 代码实现
- 定义模型结构

```
with tf.variable_scope("CNN"):
    x = tf.placeholder(tf.float32, shape=[None, 784])
    y_ = tf.placeholder(tf.float32, shape=[None, 10])

    x_image = tf.reshape(x, [-1, 28, 28, 1])
    conv1_out = conv_and_pool(x_image, kernel_size=5, out_channel=32, scope="conv1")
    conv2_out = conv_and_pool(conv1_out, kernel_size=5, out_channel=64, scope="conv2")

    conv2_out_flat = tf.reshape(conv2_out, [-1, 7 * 7 * 64])
    keep_prob = tf.placeholder(tf.float32)

    fc1 = fc_layer(conv2_out_flat, out_dim=1024, scope="fc1")
    fc1_drop = tf.nn.dropout(fc1, keep_prob=keep_prob)
    fc2 = fc_layer(fc1_drop, out_dim=10, scope="fc2")
```

CNN

- TF 代码实现
- dropout
  - 训练时随机扔掉一部分神经元，剩下神经元的输出扩大相应倍数
  - 相当于每次随机训练一个子网络，多模型集成增强性能



CNN

- TF 代码实现
- 定义损失函数、正确率等指标以及优化器

```
cross_entropy = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=fc2, labels=y_))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(fc2, 1), tf.argmax(y_, 1))
n_correct = tf.reduce_sum(tf.cast(correct_prediction, tf.float32))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
```

CNN

- TF 代码实现
- 开始训练，每 100 个 batch 在验证集上测一下

```
sess = tf.InteractiveSession()
sess.run(tf.global_variables_initializer())

t1 = time.time()
for i in range(20000):
    batch = mnist.train.next_batch(50)
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
    if i % 100 == 0:
        t2 = time.time()
        batch = mnist.validation.next_batch(50)
        train_accuracy = accuracy.eval(feed_dict={
            x: batch[0], y_: batch[1], keep_prob: 1.0})
        print("step %d, training accuracy %g, time eclipsed %g" % (i, train_accuracy, t2 - t1))
        t1 = time.time()
```

CNN

- TF 代码实现
- 训练完毕后测试

```
total_correct = 0.0
for i in range(200):
    batch = mnist.test.next_batch(50)
    total_correct += n_correct.eval(feed_dict={x: batch[0], y_: batch[1], keep_prob: 1.0})
print("test accuracy %g" % (total_correct/10000))
```

CNN

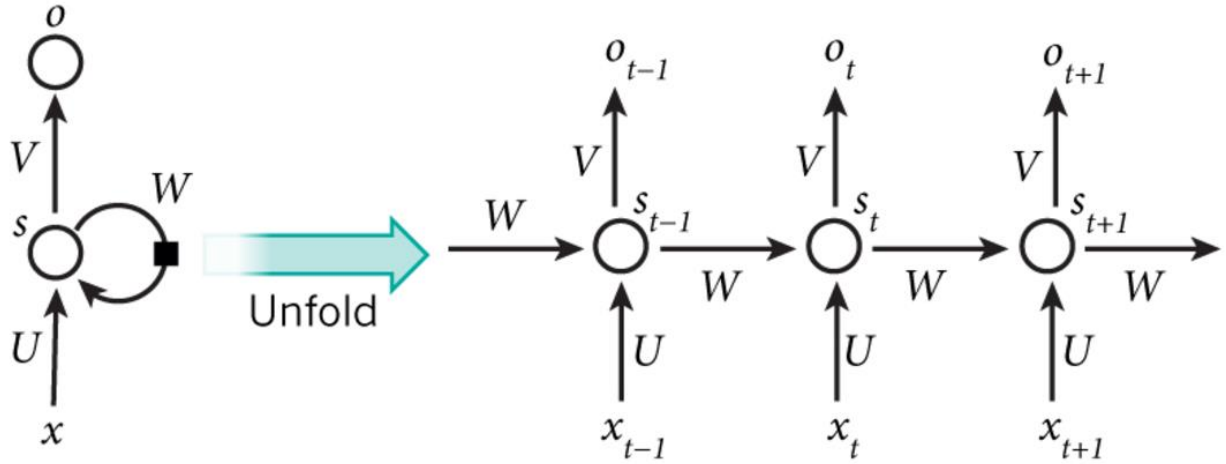
- TF 代码实现
  - 最终测试集上的分类准确率应该达到 99.2% 左右
  - 模型开始训练时有一段热身的时间，看到准确率不增长不要慌，多等等

CNN

- 其他
  - 预处理
    - 注意图像要归一化到 [0, 1] （或[-1, 1]）之间，而不是原始的灰度值 [0, 255]
  - Slim
    - 其他贡献者给 tf 封装的一套很简洁的 CNN 相关的 API
    - 之前讲的那些辅助函数其实都不用写，slim 里有现成的
      - 只需 import tensorflow.contrib.slim as slim
    - 详见 https://github.com/tensorflow/tensorflow/blob/master/tensorflow/contrib/slim/README.md
  - BN(Batch Norm)
    - 大型网络一般需要加这个
    - 如果网络有七八层或者更深，加 BN 收敛会显著加快

RNN

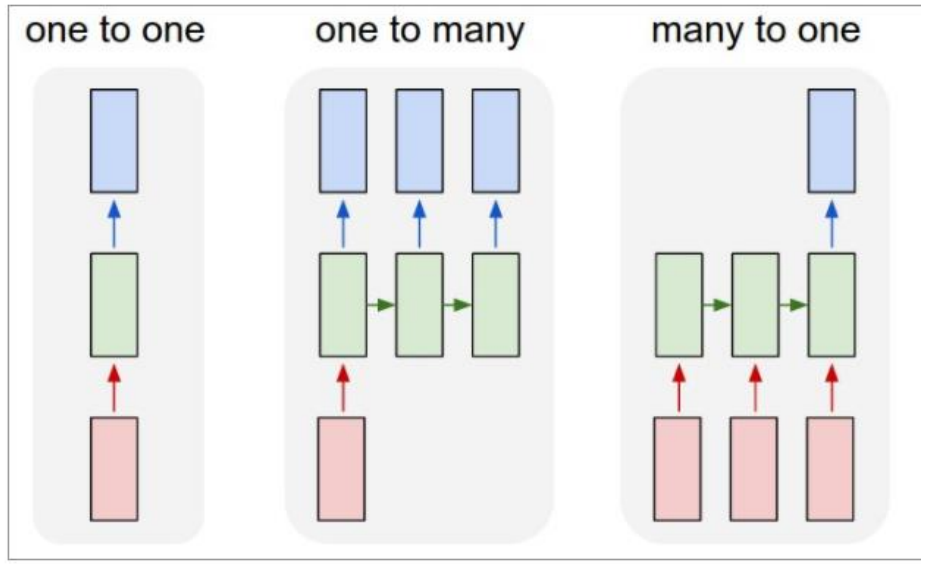
- 同样的权重在各个时间步共享
  - 理论上可以适用于任意长度的序列（例如不同的句子里单词数不同）
  - 实践中会把所有序列填充（pad）到相同长度，用掩码把无效位置挡住
    - 为了充分利用计算资源，一批样本（一个 batch）一起训练

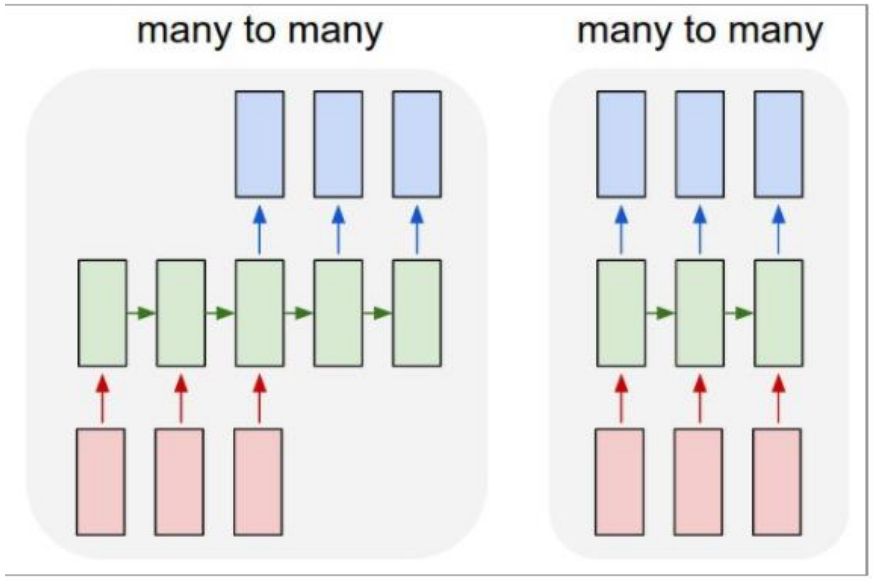


- *Denny Britz, http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/*

RNN

- RNN 的用法
  - 一对一：普通 MLP
  - 一对多：看图说话（image captioning）
  - 多对一：序列分类（如情感分析）
  - 多种多
    - 对齐：序列标注（如词性标注）
    - 不对齐：seq2seq（如机器翻译）

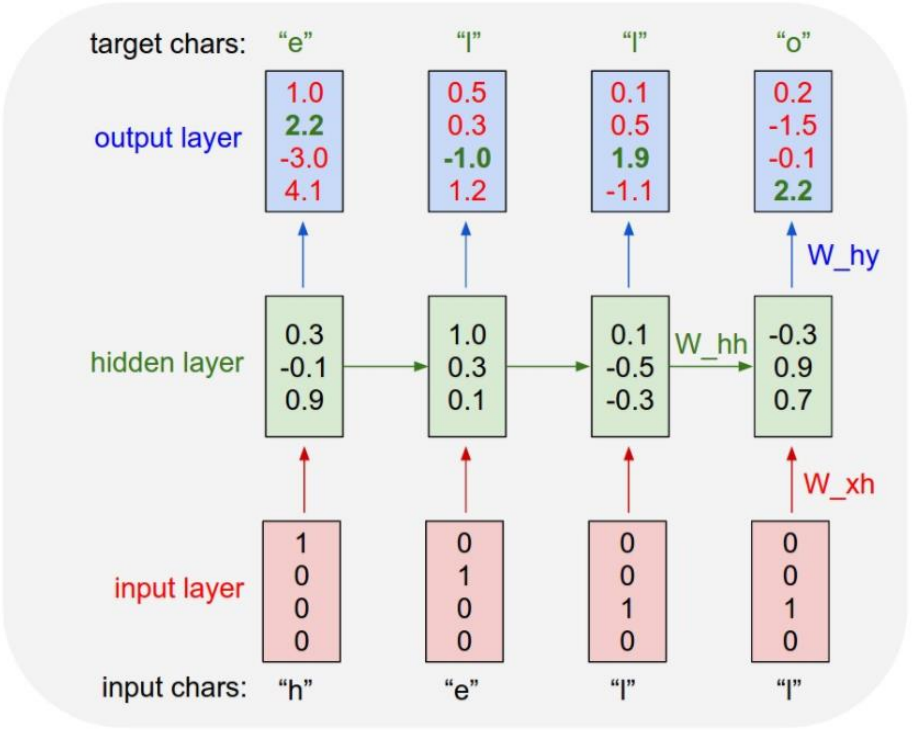




  - 图片来源
    - *Andrej Karpathy,* http://karpathy.github.io/2015/05/21/rnn-effectiveness/
    - 本文非常值得一读，后面字符语言模型的图示也出自这篇文章

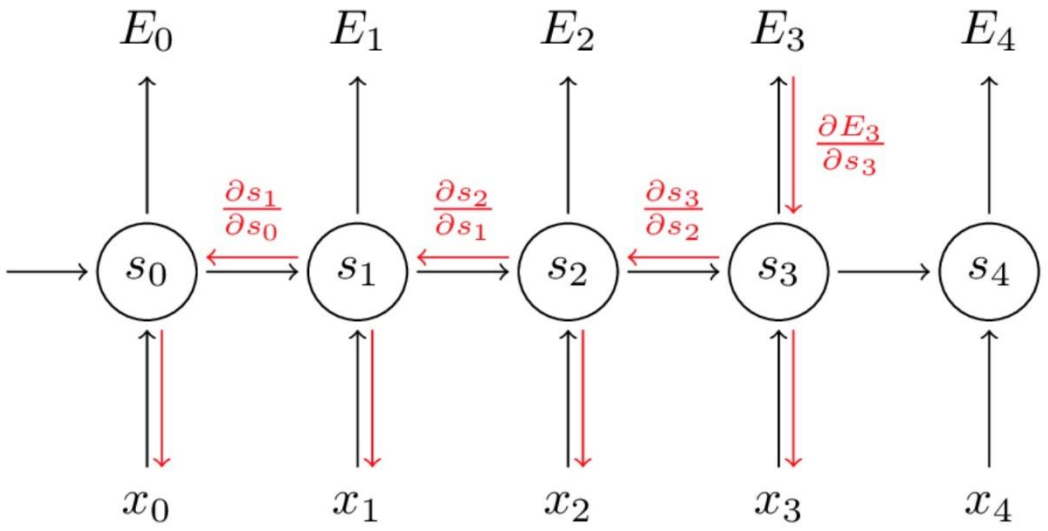
RNN

- 字符语言模型（多对多：对齐）
  - 给定前面的字符，预测下一个字符的分布
  - 目标函数：最大似然（每一步的概率相乘）



RNN

- 优化原理：BPTT（Back Propagation Through Time）
  - 如果序列过长或是无限长，会使用 Truncated BPTT
  - 实践中 RNN 处理的序列长度一般不会超过 100 步
    - 过长的序列直接丢弃或者截断
    - 语音识别中序列会长一些，有几百到上千步的



RNN

- 字符语言模型的实现
  - 根据 Udacity 深度学习课程中的例子改编
  - 源地址见 https://github.com/tensorflow/tensorflow/blob/master/tensorflow/examples/udacity/6_lstm.ipynb

RNN

- 字符语言模型的实现
  - 使用 text8 语料
    - 维基百科清洗后的前 10^8 个字符
    - 全部转化成小写字母，拼接成一行

```
 anarchism originated as a term of abuse first used against early working class
radicals including the diggers of the english revolution and the sans culottes o
f the french revolution whilst the term is still used in a pejorative way to des
cribe any act that used violent means to destroy the organization of society it
has also been taken up as a positive label by self defined anarchists the word a
narchism is derived from the greek without archons ruler chief king anarchism as
 a political philosophy is the belief that rulers are unnecessary and should be
abolished although there are differing interpretations of what this means anarch
ism also refers to related social movements that advocate the elimination of aut
horitarian institutions particularly the state the word anarchy as most anarchis
ts use it does not imply chaos nihilism or anomie but rather a harmonious anti a
uthoritarian society in place of what are regarded as authoritarian political st
```

RNN

- 字符语言模型的实现
  - 如果本地没有数据则从网络上下载

```
def maybe_download(filename='text8.zip', expected_bytes=31344016,
                   default_rul='http://mattmahoney.net/dc/'):
  """Download a file if not present, and make sure it's the right size."""
  if not os.path.exists(filename):
    filename, _ = urlretrieve(default_rul + filename, filename)
  statinfo = os.stat(filename)
  if statinfo.st_size == expected_bytes:
    print('Found and verified %s' % filename)
  else:
    print(statinfo.st_size)
    raise Exception(
      'Failed to verify ' + filename + '. Can you get to it with a browser?')
  return filename
```

RNN

- 字符语言模型的实现
  - 使用 zipfile 包直接从压缩包中读数据，不必解压

```
def read_data(filename):
    with zipfile.ZipFile(filename) as f:
        name = f.namelist()[0]
        data = tf.compat.as_str(f.read(name))
    return data



filename = maybe_download()
text = read_data(filename)
print('Data size %d' % len(text))
```

```
Found and verified text8.zip
Data size 100000000
```

RNN

- 字符语言模型的实现
  - 取前 1000 个字节做验证集

```
valid_size = 1000
valid_text = text[:valid_size]
train_text = text[valid_size:]
train_size = len(train_text)
print(train_size, train_text[:64])
print(valid_size, valid_text[:64])
```

  - 训练集和验证集各输出 64 个字符查看一下

```
99999000 ons anarchists advocate social relations based upon voluntary as
1000  anarchism originated as a term of abuse first used against earl
```

RNN

- 字符语言模型的实现
  - 词汇表大小为 27

```
vocabulary_size = len(string.ascii_lowercase) + 1  # [a-z] + ' '
first_letter = ord(string.ascii_lowercase[0])
```

  - 建立字符和 ID 的映射关系

```
def char2id(char):
    if char in string.ascii_lowercase:
        return ord(char) - first_letter + 1
    elif char == ' ':
        return 0
    else:
        print('Unexpected character: %s' % char)
        return 0
```

```
def id2char(dictid):
    if dictid > 0:
        return chr(dictid + first_letter - 1)
    else:
        return ' '
```

RNN

- 字符语言模型的实现
  - 查看映射关系

```
print(char2id('a'), char2id('z'), char2id(' '), char2id('ï'))
print(id2char(1), id2char(26), id2char(0))
```

  - 输出结果（空格的 ID 为 0）

```
1 26 0 0
a z
```

RNN

- 字符语言模型的实现
  - 写一个类用于生成数据
    - 字符流（近乎）无限长，每 10 步一截断，作为一个样本
    - 批大小为 64，即每 64 个样本一起训练

```
batch_size = 64
num_unrollings = 10


class BatchGenerator(object):
    def __init__(self, text, batch_size, num_unrollings):
        self._text = text
        self._text_size = len(text)
        self._batch_size = batch_size
        self._num_unrollings = num_unrollings
        segment = self._text_size // batch_size
        self._cursor = [offset * segment for offset in range(batch_size)]
        self._last_batch = self._next_batch()
```

RNN

- 字符语言模型的实现
  - 数据生成算法
    - 把整个字符流切分成 64 段，在每一段里用一个游标指示当前位置
    - 游标是包含 64 个元素的列表
    - 开始时 64 个游标分别指向 64 个字符段的开始位置
    - 然后从每个游标处各取 1 个字符，形成 64 个样本的首字符
    - 重复以上步骤 10 次，得到 64 个长度为 10的样本
    - 将这 64 个样本打包送给模型训练
    - 无限重复以上过程
      - 若游标到达字符流的末尾则回到开头处重新开始

RNN

- 字符语言模型的实现
  - 给每个样本各取一个字符
    - 此处字符用 one-hot 方法表示

```
def _next_batch(self):
    """Generate a single batch from the current cursor position in the data."""
    batch = np.zeros(shape=(self._batch_size, vocabulary_size), dtype=np.float)
    for b in range(self._batch_size):
        batch[b, char2id(self._text[self._cursor[b]])] = 1.0
        self._cursor[b] = (self._cursor[b] + 1) % self._text_size
    return batch
```

RNN

- 字符语言模型的实现
  - 调用 10 次 _next_batch()，得到长度为 10+1 的 64 个样本
    - 返回值是一个长为 10+1 的列表，其中每个元素是形状为 [64, 27] 的数组
      - 不考虑类型问题的话，可以看成是形状为 [10+1, 64, 27] 的数组

```
def next(self):
    """Generate the next array of batches from the data. The array consists of
    the last batch of the previous array, followed by num_unrollings new ones.
    """
    batches = [self._last_batch]
    for step in range(self._num_unrollings):
        batches.append(self._next_batch())
    self._last_batch = batches[-1]
    return batches
```

RNN

- 字符语言模型的实现
  - 为什么序列长度是 10+1
    - 因为输入序列和目标序列错位一步
    - 需要生成长为 11 的片段，然后取前 10 个字符作为输入，后 10 个字符作为标签
  - RNN 的输入通常有两种顺序：[T, B, D] 或 [B, T, D]
    - T/B/D 分别代表时间步数/批大小/输入维度
    - 前者适合沿时间步循环
      - 例如在 tf.while_loop 中应用
    - 后者适合展示数据
      - 因为 B 取一个定值时，切片得到的 [T, D] 数组里都是跟该样本相关的值
    - 这里使用的是前一种

RNN

- 字符语言模型的实现
  - 把一个 batch 的 ID 转化成字符序列
    - probabilities 是形如 [B, D] 的 numpy array
    - 返回值是长为 B 的字符列表

```
def characters(probabilities):
    """Turn a 1-hot encoding or a probability distribution over the possible
    characters back into its (most likely) character representation."""
    return [id2char(c) for c in np.argmax(probabilities, 1)]
```

RNN

- 字符语言模型的实现
  - 把一个 batch 的 ID 转化成字符序列
    - batches 是前面的 next() 函数的返回值
      - 长为 10+1 的列表，其中每个元素是形状为 [64, 27] 的数组
    - 对其中每个元素应用 characters() 函数，并对结果进行拼接
      - s 是长为 64 的列表，其中每个元素是一个长为 10+1 的字符串

```
def batches2string(batches):
    """Convert a sequence of batches back into their (most likely) string
    representation."""
    s = [''] * batches[0].shape[0]
    for b in batches:
        s = [''.join(x) for x in zip(s, characters(b))]
    return s
```

RNN

- 字符语言模型的实现
  - 输出一些 batches 检查一下

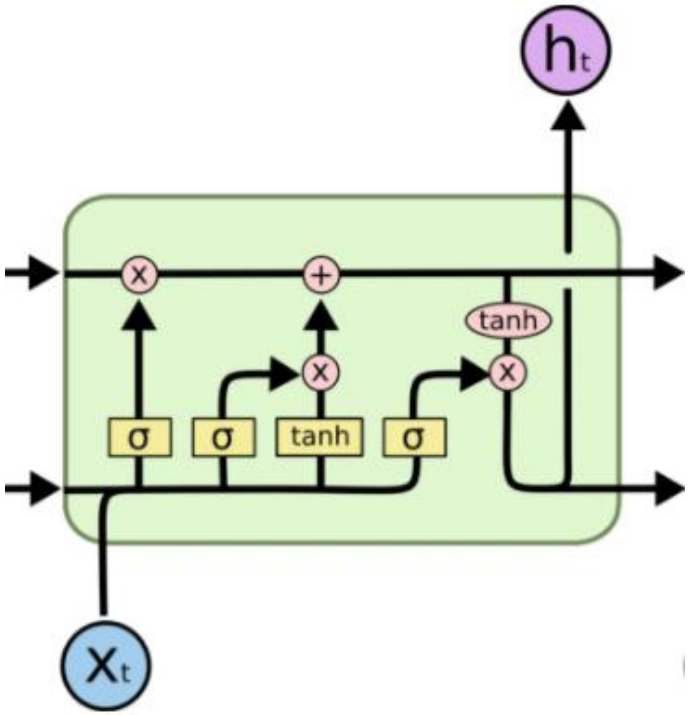
```
train_batches = BatchGenerator(train_text, batch_size, num_unrollings)
valid_batches = BatchGenerator(valid_text, 1, 1)

print(batches2string(train_batches.next()))
print(batches2string(train_batches.next()))
print(batches2string(valid_batches.next()))
print(batches2string(valid_batches.next()))
```

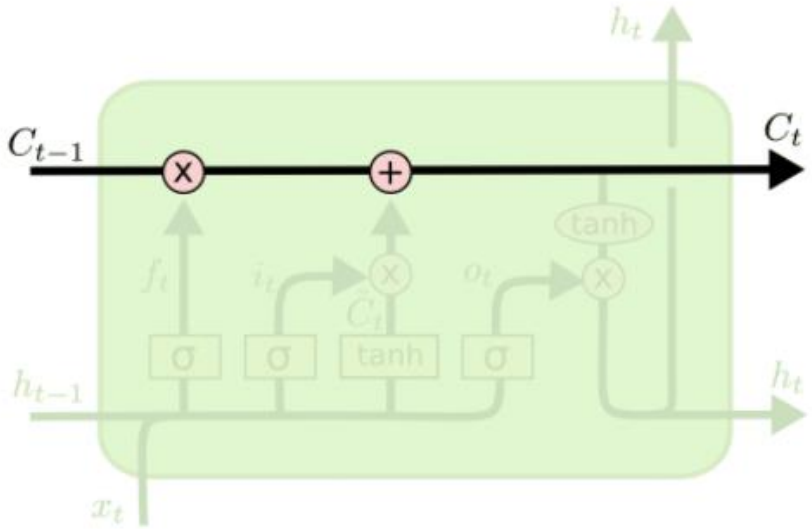
  - 前两行应该有 64 个元素，这里放不下

```
['ons anarchi', 'when milita', 'lleria arch', ' abbeys and', 'married urr',
['ists advoca', 'ary governm', 'hes nationa', 'd monasteri', 'raca prince',
[' a']
['an']
```

RNN

- 字符语言模型的实现
  - LSTM 单元





$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

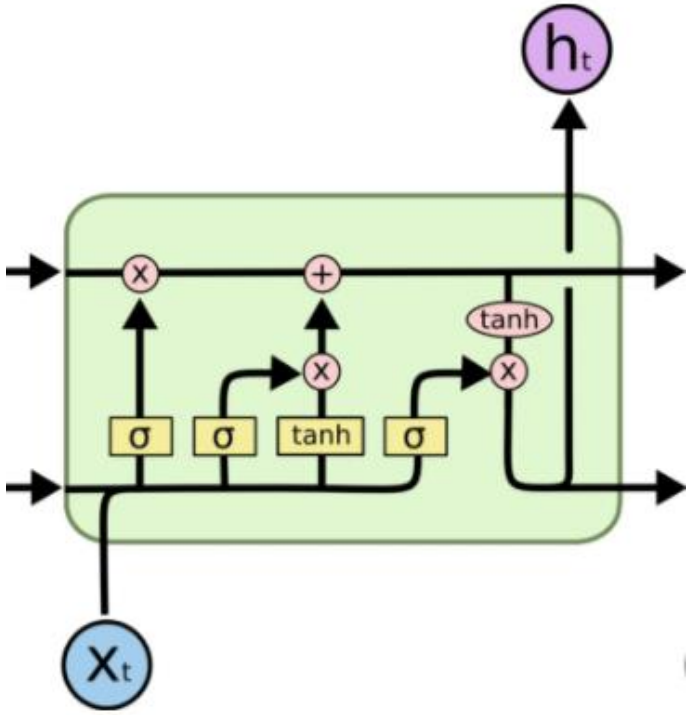
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

$$o_t = \sigma\left(W_o \; [h_{t-1}, x_t] \; + \; b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

RNN

- 字符语言模型的实现
  - 注意区分 RNN Cell 的状态和输出
    - 状态：在各个时间步之间传递的变量
    - 输出：供模型下一层使用的变量
  - 对单层 vanilla RNN 和 GRU 来说
    - 输出 = 状态
  - 对单层 LSTM 来说
    - 输出：h
    - 状态：(c, h) tuple
  - 多层 RNN Cell 的情形
    - 详见 https://zhuanlan.zhihu.com/p/28919765



RNN

- 字符语言模型的实现
  - LSTM 单元的定义
    - 各个门相关的变量

```
num_nodes = 64

graph = tf.Graph()
with graph.as_default():
    # Parameters:
    # Input gate: input, previous output, and bias.
    ix = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes], -0.1, 0.1))
    im = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
    ib = tf.Variable(tf.zeros([1, num_nodes]))
    # Forget gate: input, previous output, and bias.
    fx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes], -0.1, 0.1))
    fm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
    fb = tf.Variable(tf.zeros([1, num_nodes]))
```

RNN

- 字符语言模型的实现
  - LSTM 单元的定义
    - 各个门相关的变量

```
# Memory cell: input, state and bias.
cx = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes], -0.1, 0.1))
cm = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
cb = tf.Variable(tf.zeros([1, num_nodes]))
# Output gate: input, previous output, and bias.
ox = tf.Variable(tf.truncated_normal([vocabulary_size, num_nodes], -0.1, 0.1))
om = tf.Variable(tf.truncated_normal([num_nodes, num_nodes], -0.1, 0.1))
ob = tf.Variable(tf.zeros([1, num_nodes]))
```

RNN

- 字符语言模型的实现
  - LSTM 单元的定义
    - 内部计算过程
      - 这里 i 是当前时间步的输入（一般记作 x_t）
      - o 是前一时间步的输出（一般记作 h_{t-1}）
      - state 是前一时间步的细胞状态（一般记作 c_{t-1}）

```
# Definition of the cell computation.
def lstm_cell(i, o, state):
    """Create a LSTM cell. See e.g.: http://arxiv.org/pdf/1402.1128v1.pdf
    Note that in this formulation, we omit the various connections between the
    previous state and the gates."""
    input_gate = tf.sigmoid(tf.matmul(i, ix) + tf.matmul(o, im) + ib)
    forget_gate = tf.sigmoid(tf.matmul(i, fx) + tf.matmul(o, fm) + fb)
    update = tf.matmul(i, cx) + tf.matmul(o, cm) + cb
    state = forget_gate * state + input_gate * tf.tanh(update)
    output_gate = tf.sigmoid(tf.matmul(i, ox) + tf.matmul(o, om) + ob)
    return output_gate * tf.tanh(state), state
```

RNN

- 字符语言模型的实现
  - 模型定义
    - 输入和输出序列的占位符

```
# Input data.
train_data = list()
for _ in range(num_unrollings + 1):
    train_data.append(
        tf.placeholder(tf.float32, shape=[batch_size, vocabulary_size]))
train_inputs = train_data[:num_unrollings]
train_labels = train_data[1:]  # labels are inputs shifted by one time step.
```

RNN

- 字符语言模型的实现
  - LSTM 单元的定义
    - 接到输出分类器的变量和前一个 batch 的输入和状态

```
# Variables saving state across unrollings.
saved_output = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)
saved_state = tf.Variable(tf.zeros([batch_size, num_nodes]), trainable=False)
# Classifier weights and biases.
w = tf.Variable(tf.truncated_normal([num_nodes, vocabulary_size], -0.1, 0.1))
b = tf.Variable(tf.zeros([vocabulary_size]))
```

RNN

- 字符语言模型的实现
  - 模型定义
    - 在沿时间轴展开的计算图上循环，得到输出
    - RNN 的初始状态是前一个 batch 的最终状态
    - outputs 是长为 10 的列表，其中每个元素是形如 [batch_size, num_nodes] 的张量
      - 这个例子里 batch_size 和 num_nodes 都是 64

```
# Unrolled LSTM loop.
outputs = list()
output = saved_output
state = saved_state
for i in train_inputs:
    output, state = lstm_cell(i, output, state)
    outputs.append(output)
```

RNN

- 字符语言模型的实现
  - 模型定义
    - 在 batch 之间传递模型内部状态
      - saved_output 和 saved_state 记录模型处理前一批数据后得到的最终输出和状态
      - 仅截断梯度的传播，不清空模型内部状态

```
# State saving across unrollings.
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(outputs, 0), w, b)
    loss = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(
            labels=tf.concat(train_labels, 0), logits=logits))
```

RNN

- 字符语言模型的实现
  - 模型定义
    - 用 tf.control_dependencies 来控制运算顺序
      - 要求在计算 logits 和 loss 前，必须把最终的 output 和 state 保存进 saved_output 和 saved_state 里，防止下一个 batch 出错

```
# State saving across unrollings.
with tf.control_dependencies([saved_output.assign(output),
                              saved_state.assign(state)]):
    # Classifier.
    logits = tf.nn.xw_plus_b(tf.concat(outputs, 0), w, b)
    loss = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(
            labels=tf.concat(train_labels, 0), logits=logits))
```

RNN

- 字符语言模型的实现
  - 关于运算顺序
    - CNN 一般不存在这个问题
    - 对于这个例子，另一个解决方案是把模型初始状态定义成 placeholder
      - 每个 batch 求值后都把最终状态保存在 numpy 数组中
      - 下一个 batch 计算时再把保存的值 feed 给相应的 placeholder
    - TF 中还有一个 state_saving_rnn，应该也能解决这个问题（但我没用过）

RNN

- 字符语言模型的实现
  - 模型定义：logits 的计算
    - outputs 是长为 T=10 的列表，其中每个元素是形如 [batch_size, num_nodes] 的张量
    - tf.concat(outputs, 0) 是把 outputs 中的所有张量沿着第 0 个轴拼起来
      - 结果是形如 [B * T, num_nodes] 的张量
      - 每 B 行对应一个时间步
        - 前 B 行对应的是这个 batch 里各个样本第 1 个时间步的输出
        - 第 B+1 行到第 2B 行对应的是这个 batch 里各个样本第 2 个时间步的输出
        - 以此类推
    - 然后做线性变换得到的 logits 是形如 [B*T, vocab_size]的张量

```
# Classifier.
logits = tf.nn.xw_plus_b(tf.concat(outputs, 0), w, b)
```

RNN

- 字符语言模型的实现
  - 模型定义：loss 的计算
    - train_labels 是长为 T=10 的列表，其中每个元素是形如 [B, vocab_size] 的占位符
      - 回想：占位符也是特殊的张量
    - tf.concat(train_labels, 0) 得到一个形如 [B * T, vocab_size] 的张量
      - 同样，每 B 行对应一个时间步
    - 此时 logits 和 label 形状相同（都是 [B * T, vocab_size]），每一行也都是对应的
    - tf.nn.softmax_cross_entropy_with_logits() 会自动对 logits 的最后一维做 softmax
      - 得到形如 [B*T] 的一维张量，就是这一批数据里每一个时间步的交叉熵
    - 最后用 tf.reduce_mean() 来得到均值 loss

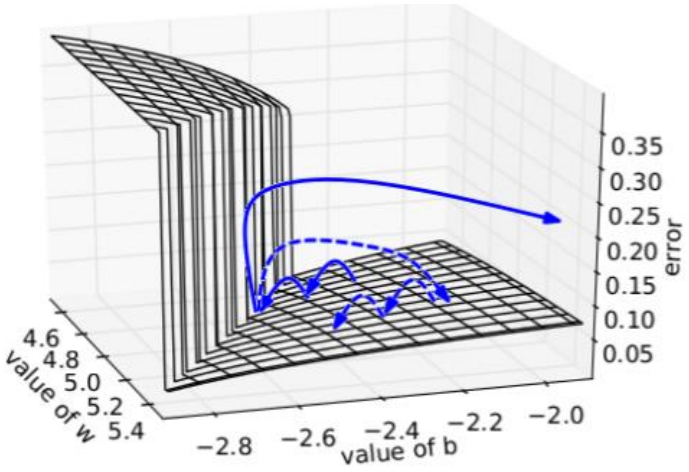
```
    loss = tf.reduce_mean(
        tf.nn.softmax_cross_entropy_with_logits(
            labels=tf.concat(train_labels, 0), logits=logits))
```

RNN

- 字符语言模型的实现
  - 优化器
    - global_step 用于统计模型训练了多少个 batch
    - 学习率采用指数衰减
    - 优化器用最普通的梯度下降

```
# Optimizer.
global_step = tf.Variable(0)
learning_rate = tf.train.exponential_decay(
    10.0, global_step, 5000, 0.1, staircase=True)
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)
```

RNN



- 字符语言模型的实现
  - 梯度裁减
    - 这里没有直接调用 optimizer.minimize
    - 而是先用 compute_gradients 求导，再做 gradient clipping，最后 apply_gradients
    - 解决 RNN 的梯度爆炸问题
      - *Razvan Pascanu et al.* ***On the difficulty of training Recurrent Neural Networks****. ICML'12*

```
# Optimizer.
global_step = tf.Variable(0)
learning_rate = tf.train.exponential_decay(
    10.0, global_step, 5000, 0.1, staircase=True)
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
gradients, v = zip(*optimizer.compute_gradients(loss))
gradients, _ = tf.clip_by_global_norm(gradients, 1.25)
optimizer = optimizer.apply_gradients(
    zip(gradients, v), global_step=global_step)
```

RNN

- 字符语言模型的实现
  - 定义一个新模型用于生成新文本
    - 模型每次运行一步，预测下一个字符的分布
    - 然后从分布中采样出一个字符，再进行一步预测
    - 如此循环直到生成一大段文本
  - 只定义除模型参数以外的变量
    - 例如用来在不同 batch 间传递模型状态的变量、占位符等
    - 模型参数复用先前定义的、用于训练的模型的参数

RNN

- 字符语言模型的实现
  - tf.group()：把多个 Op 打包成一个 Op

```
# Predictions.
train_prediction = tf.nn.softmax(logits)

# Sampling and validation eval: batch 1, no unrolling.
sample_input = tf.placeholder(tf.float32, shape=[1, vocabulary_size])
saved_sample_output = tf.Variable(tf.zeros([1, num_nodes]))
saved_sample_state = tf.Variable(tf.zeros([1, num_nodes]))
reset_sample_state = tf.group(
    saved_sample_output.assign(tf.zeros([1, num_nodes])),
    saved_sample_state.assign(tf.zeros([1, num_nodes])))
sample_output, sample_state = lstm_cell(
    sample_input, saved_sample_output, saved_sample_state)
with tf.control_dependencies([saved_sample_output.assign(sample_output),
                              saved_sample_state.assign(sample_state)]):
    sample_prediction = tf.nn.softmax(tf.nn.xw_plus_b(sample_output, w, b))
```

RNN

- 字符语言模型的实现
  - 其他辅助函数
    - logprob
      - predictions 是形如 [None, vocab_size] 的二维数组，每行代表一个字符集上的概率分布
        - None 的意思是任意数值均可
      - labels 是与之同型的 one-hot 的数组：每行有且仅有一个 1，其余都是 0
      - 先对过小的值做一下处理，避免取对数时数值下溢
      - 每一行都算一下真正的标签的负对数，然后取平均

```
def logprob(predictions, labels):
  """Log-probability of the true labels in a predicted batch."""
  predictions[predictions < 1e-10] = 1e-10
  return np.sum(np.multiply(labels, -np.log(predictions))) / labels.shape[0]
```

RNN

- 字符语言模型的实现
  - 其他辅助函数
    - sample_distribution
      - 从 unigram distribution 中采样
      - distribution 是一个一维数组（或列表），并且代表了一个概率分布

```
def sample_distribution(distribution):
  """Sample one element from a distribution assumed to be an array of normalized
  probabilities.
  """
  r = random.uniform(0, 1)
  s = 0
  for i in range(len(distribution)):
    s += distribution[i]
    if s >= r:
      return i
  return len(distribution) - 1
```

RNN

- 字符语言模型的实现
  - 其他辅助函数
    - sample：
      - 从 prediction（形如 [None, vocab_size] 的二维数组）的第一行代表的概率分布中采样
      - 表示成一个 one-hot 的分布 p，以列向量的形式返回

```
def sample(prediction):
  """Turn a (column) prediction into 1-hot encoded samples."""
  p = np.zeros(shape=[1, vocabulary_size], dtype=np.float)
  p[0, sample_distribution(prediction[0])] = 1.0
  return p
```

RNN

- 字符语言模型的实现
  - 其他辅助函数
    - random_distribution：
      - 随机生成一个长度为 vocab_size 的分布

```
def random_distribution():
  """Generate a random column of probabilities."""
  b = np.random.uniform(0.0, 1.0, size=[1, vocabulary_size])
  return b/np.sum(b, 1)[:,None]
```

RNN

- 字符语言模型的实现
  - 训练过程
    - 共计训练 7001 步
    - 每 100 步输出一些统计信息

```
num_steps = 7001
summary_frequency = 100

with tf.Session(graph=graph) as session:
    tf.global_variables_initializer().run()
    print('Initialized')
    mean_loss = 0
    for step in range(num_steps):
        batches = train_batches.next()
        feed_dict = dict()
        for i in range(num_unrollings + 1):
            feed_dict[train_data[i]] = batches[i]
        _, l, predictions, lr = session.run(
            [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
        mean_loss += l
```

RNN

- 字符语言模型的实现
  - 训练过程
    - 反复读取数据、生成训练样本、执行梯度下降的 Op 即可
    - 也可以对其他 Op 求值，打印一些信息帮助判断是否收敛

```
for step in range(num_steps):
    batches = train_batches.next()
    feed_dict = dict()
    for i in range(num_unrollings + 1):
        feed_dict[train_data[i]] = batches[i]
    _, l, predictions, lr = session.run(
        [optimizer, loss, train_prediction, learning_rate], feed_dict=feed_dict)
```

RNN

- 字符语言模型的实现
  - 训练过程
    - 每 100 步输出平均 loss 查看一下

```
if step % summary_frequency == 0:
    if step > 0:
        mean_loss = mean_loss / summary_frequency
    # The mean loss is an estimate of the loss over the last few batches.
    print(
        'Average loss at step %d: %f learning rate: %f' % (step, mean_loss, lr))
    mean_loss = 0
    labels = np.concatenate(list(batches)[1:])
    print('Minibatch perplexity: %.2f' % float(
        np.exp(logprob(predictions, labels))))
```

RNN

- 字符语言模型的实现
  - 训练过程
    - 每 1k 步随机生成 5 段长为 80 的字符序列
    - 先生成一个字符，然后从该字符开始生成长为 79 的字符序列

```
if step % (summary_frequency * 10) == 0:
    # Generate some samples.
    print('=' * 80)
    for _ in range(5):
        feed = sample(random_distribution())
        sentence = characters(feed)[0]
        reset_sample_state.run()
        for _ in range(79):
            prediction = sample_prediction.eval({sample_input: feed})
            feed = sample(prediction)
            sentence += characters(feed)[0]
        print(sentence)
    print('=' * 80)
```

RNN

- 字符语言模型的实现
  - 训练过程
    - 计算验证集上的困惑度
      - 困惑度：平均负对数似然，越小越好
      - 物理意义：猜下一个词（这里是字母）时模型会在几个选项中犹豫不决

```
# Measure validation set perplexity.
reset_sample_state.run()
valid_logprob = 0
for _ in range(valid_size):
    b = valid_batches.next()
    predictions = sample_prediction.eval({sample_input: b[0]})
    valid_logprob = valid_logprob + logprob(predictions, b[1])
print('Validation set perplexity: %.2f' % float(np.exp(
    valid_logprob / valid_size)))
```

RNN

- 字符语言模型的实现
  - 运行结果
    - step 0：模型什么都不知道，在 27 个字符里瞎猜

```
Average loss at step 0: 3.298197 learning rate: 10.000000
Minibatch perplexity: 27.06
================================================================================
yalglet  tiggeiwixonss cmle  ilrsrfkyo tbzt ol qpy cvie cpbtravvahojohl xtdnfihy
katxm ec mxzfb nssgegtaqbalhajqs  fe  rclsdpt sfvbshxrvr  oal rzvntwotmaefowhkak
a xbmnptixhlepilxtu cxaddejnic gietlmlnkepserirnmieq hslszeagcjynjtx oxup sx daz
ekrcibtpz vcnpvagrjcmrbx hngbrscwrvewfzue  nfqtchmsjuibhpgoatt ltrdi t cebucgccv
dtdiei byzraxi quigw bzu ty laurie   thmbak fsmr bpfvayyca ci tn ber slxou wuarx
================================================================================
```

RNN

- 字符语言模型的实现
  - 运行结果
    - 之后 loss 和 ppl 迅速下降

```
Validation set perplexity: 20.31
Average loss at step 100: 2.589565 learning rate: 10.000000
Minibatch perplexity: 11.15
Validation set perplexity: 10.41
Average loss at step 200: 2.251975 learning rate: 10.000000
Minibatch perplexity: 8.66
Validation set perplexity: 8.55
Average loss at step 300: 2.094883 learning rate: 10.000000
Minibatch perplexity: 7.42
Validation set perplexity: 7.98
Average loss at step 400: 1.996019 learning rate: 10.000000
Minibatch perplexity: 7.53
```

RNN

- 字符语言模型的实现
  - 运行结果
    - step 1k：模型学到了一些短词，例如 of/the/to/serve/count

```
Average loss at step 1000: 1.817202 learning rate: 10.000000
Minibatch perplexity: 5.49
================================================================================
 nunger at omer eter commutere ecciouting mage to the che wellibularysed a therm
ther to feill to difcent uroou is of daughtion any his nor the centurn internafe
dies the rebuliva or elentry of proplehiciun c flect to tre thes the preses bo b
x reformation hinct of the rudition live comiem for menes s begn count the serve
d anverted enriced century boigred are weer the meich diftereniylegier diever de
================================================================================
```

RNN

- 字符语言模型的实现
  - 运行结果
    - step 7k：更像英语了，最终 valid ppl = 4.27
    - 模型如果更大一些、训练时间更长一些，效果应该会更好，长词也能学出来

```
Average loss at step 7000: 1.576163 learning rate: 1.000000
Minibatch perplexity: 5.03
================================================================================
pendation as the miscrepie scier leber the computes effcanureary entides and boi
le proboter cartina sumerian usas becaused byst struld aramic vansorth d empersy
 fracterimate is intervers theory borki sersh four common heryda kings and some
bused that the paint is or the can forming that how deatis quanttent propriment
s was longally chettorstpends lock pailization or the wook country mact of jans
================================================================================
Validation set perplexity: 4.27
```

RNN

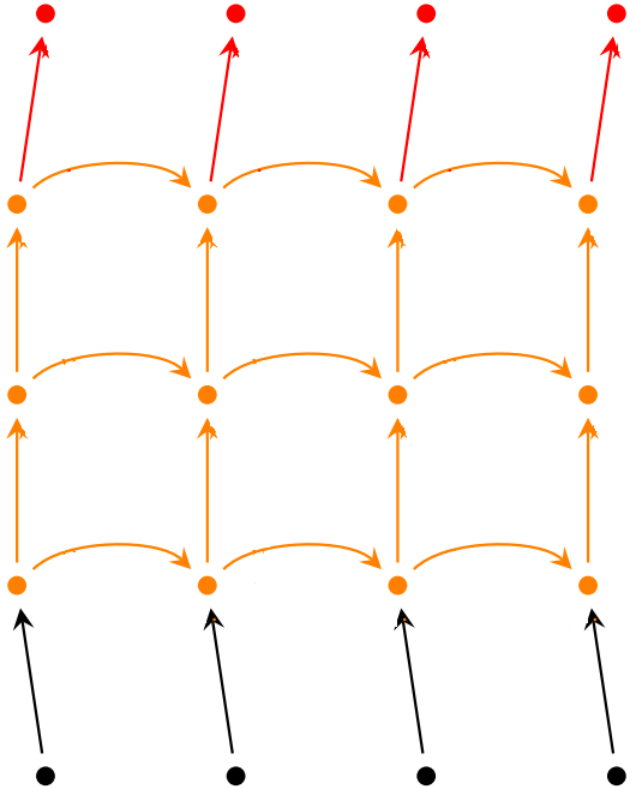
- TF 中的 RNN API
  - RNNCell
    - 详见 https://www.tensorflow.org/versions/master/api_guides/python/contrib.rnn

Core RNN Cells for use with TensorFlow's core RNN methods

- `tf.contrib.rnn.BasicRNNCell`
- `tf.contrib.rnn.BasicLSTMCell`
- `tf.contrib.rnn.GRUCell`
- `tf.contrib.rnn.LSTMCell`
- `tf.contrib.rnn.LayerNormBasicLSTMCell`

RNN

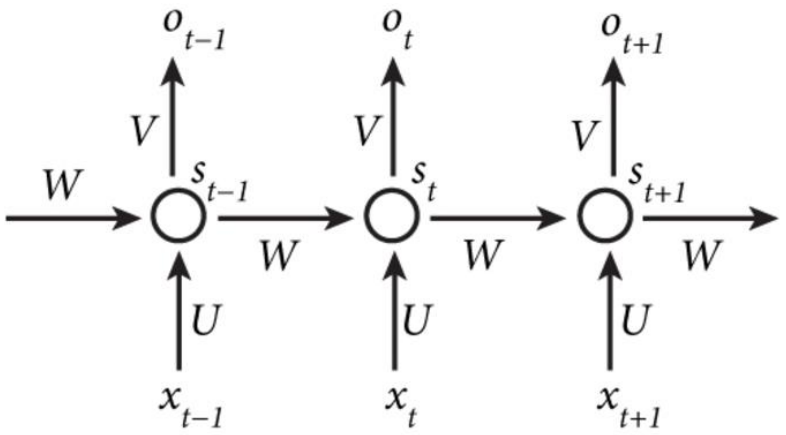
- TF 中的 RNN API
  - MultiRNNCell
    - 接受一个 RNNCell 的列表，把它们封装起来
    - 在外界看来，整体好像也是一个普通的 RNNCell
    - 用于定义多层 RNN



- 图片修改自 *Denny Britz, http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-2-implementing-a-language-model-rnn-with-python-numpy-and-theano/*
- 实现详见 https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/ops/rnn_cell_impl.py

RNN

- TF 中的 RNN API
  - RNNCell 的本质
    - 其实在 TF 里，RNNCell 是一个抽象类，常用的 LSTM 等都是它的子类
    - RNNCell 最重要的方法是 __call__
      - 给定一个输入和前一步的状态
      - 返回一个输出和更新后的状态
      - $o_t, s_t = call(x_t, s_{t-1})$
    - 还有一个方法是 zero_state
      - 返回零状态



RNN

Aliases:

- `tf.contrib.rnn.static_rnn`
- `tf.nn.static_rnn`

```
tf.nn.static_rnn(
    cell,
    inputs,
    initial_state=None,
    dtype=None,
    sequence_length=None,
    scope=None
)
```

- TF 中的 RNN API
  - tf.nn.static_rnn
    - cell 是前面某种 RNNCell 的实例
    - inputs 是一个长为 T 的列表
      - 其中每个元素是形如 [B, D] 的张量
    - initial_state 是初始状态
    - sequence_length 表示这个 batch 里每个样本的实际长度
    - 返回值
      - 返回一个 tuple：(outputs, state)
      - outputs 是长为 T 的列表，每个元素是相应时间步的输出（形如 [B, layer_size] 的张量）
      - state 是 RNNCell 的末状态
    - https://www.tensorflow.org/versions/master/api_docs/python/tf/nn/static_rnn

RNN

- TF 中的 RNN API
  - tf.nn.static_rnn
    - 基本形式等价于右边的代码

```
state = cell.zero_state(...)
outputs = []
for input_ in inputs:
  output, state = cell(input_, state)
  outputs.append(output)
return (outputs, state)
```

    - 若给定序列长度则相当于下面的代码
      - 无效位置的输出用零填充
      - 无效位置的状态保留最后一个有效位置的状态

```
(output, state)(b, t) =
  (t >= sequence_length(b))
    ? (zeros(cell.output_size), states(b, sequence_length(b) - 1))
    : cell(input(b, t), state(b, t - 1))
```

RNN

- TF 中的 RNN API
  - tf.nn.dynamic_rnn
    - cell 是前面某种 RNNCell
    - inputs
      - 形如 [B, T, D] 或 [T, B, D] 的三维张量
      - 取决于参数 time_major（默认为 False）
    - 返回值
      - tuple (outputs, state)
      - 假设 D’ = cell.output_size
      - outputs 是形如 [B, T, D’] 或 [T, B, D’] 的张量
        - 取决于参数 time_major
      - state 是最终模型在该 batch 上的最终状态

tf.nn.dynamic_rnn

```
tf.nn.dynamic_rnn(
    cell,
    inputs,
    sequence_length=None,
    initial_state=None,
    dtype=None,
    parallel_iterations=None,
    swap_memory=False,
    time_major=False,
    scope=None
)
```

RNN

- TF 中的 RNN API
  - static_rnn 和 dynamic_rnn 的区别
    - 参数类型、各个维度意义不一样
    - dynamic_rnn 内部是用 tf.while_loop 实现的
      - 符号循环，输入有多少步计算时就展开多少步
      - 不同 batch 的时间步数目可以不同
    - static_rnn 预先构建好计算图
      - 展开的步数是确定的
      - 建图更费时间，显存消耗更大，速度略快于 dynamic_rnn
    - 建议多用 dynamic_rnn

RNN

- TF 中的 RNN API
  - 例一

```
# create a BasicRNNCell
rnn_cell = tf.nn.rnn_cell.BasicRNNCell(hidden_size)

# 'outputs' is a tensor of shape [batch_size, max_time, cell_state_size]

# defining initial state
initial_state = rnn_cell.zero_state(batch_size, dtype=tf.float32)

# 'state' is a tensor of shape [batch_size, cell_state_size]
outputs, state = tf.nn.dynamic_rnn(rnn_cell, input_data,
                                   initial_state=initial_state,
                                   dtype=tf.float32)
```

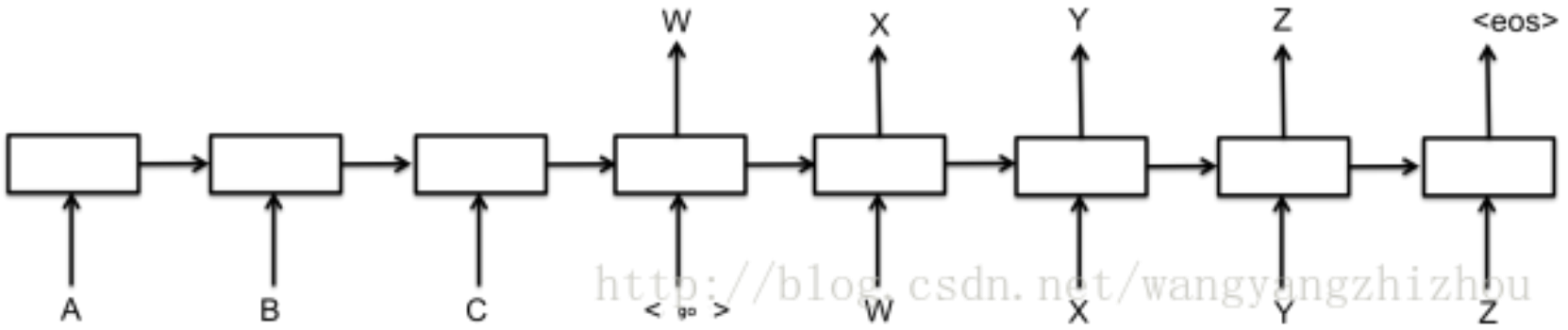
RNN

- TF 中的 RNN API
  - 例二
    - state = (s1, s2)，其中 s1 和 s2 均为 LSTMStateTuple
    - s1 = LSTMStateTuple(c=Tensor([B, 128]), h=Tensor([B, 128]))，s2 类似

```
# create 2 LSTMCells
rnn_layers = [tf.nn.rnn_cell.LSTMCell(size) for size in [128, 256]]

# create a RNN cell composed sequentially of a number of RNNCells
multi_rnn_cell = tf.nn.rnn_cell.MultiRNNCell(rnn_layers)

# 'outputs' is a tensor of shape [batch_size, max_time, 256]
# 'state' is a N-tuple where N is the number of LSTMCells containing a
# tf.contrib.rnn.LSTMStateTuple for each cell
outputs, state = tf.nn.dynamic_rnn(cell=multi_rnn_cell,
                                   inputs=data,
                                   dtype=tf.float32)
```

seq2seq

- 多对多：输入和输出都不定长
  - 用一个 RNN 读取源序列
    - 得到一个编码了整个序列语义信息的向量
  - 然后把该向量喂给另一个 RNN
    - 解码出目标序列
  - 通过最大似然来训练



seq2seq

- 本节主要使用 tf.contrib.seq2seq 中的 API
  - 因为从头写真的太麻烦了……
  - 文档见 https://www.tensorflow.org/versions/master/api_guides/python/contrib.seq2seq
  - 这一套 API 已经相对稳定了，而且很好用
  - 如果在网上看到有人用 tf.contrib.legacy_seq2seq，就不要学了
    - 这是老接口，现在已经逐渐弃用了

seq2seq

- 问题描述（toy example）
  - 删除奇数
    - 给定一个序列，例如 [2, 5, 6, 7, 8, 9, 2, 3, 1]
    - 要求返回删除奇数后得到的序列，即 [2, 6, 8, 2]
  - 生成数据
    - 为简便起见，假定所有数字都在 1~10 之间
    - 用 0 表示序列结束
    - 假定所有序列长度都不超过 24
    - 生成算法
      - 随机选取一些奇数，再随机选取一些偶数
      - 混合到一起打乱，就是输入序列
      - 删掉其中的奇数，就是输入序列

seq2seq

- 问题描述（toy example）
  - 数据样例
    - 此处 batch_size = 3
    - 所有样本都填充到和当前 batch 最长的序列相同的长度（此处是 19）
  - 代码详见 seq2seq.py 中的 generate_data()

```
Sample x:
[[ 5  1  7 10 10 10  7  2  5 10  3  8  4  6  7  8  0  0  0]
 [ 2  8  6  3  7  5  5  6 10  6  4  2  3  1  7  8  9  3  4]
 [ 9  1  3  8  3  6  2  6  0  0  0  0  0  0  0  0  0  0  0]]
Expected y:
[[10 10 10  2 10  8  4  6  8  0  0]
 [ 2  8  6  6 10  6  4  2  8  4  0]
 [ 8  6  2  6  0  0  0  0  0  0  0]]
```

seq2seq

- 注
  - 实际问题中序列长短可能会差异很大，从个位数变化到三位数
  - 这时常采取的策略是分桶
  - 例如长度在 10 以内的序列组成一个桶、长度在 [10, 20] 的序列组成另一个桶等
  - 每次取样本组成一个 batch 时，都从某一个桶里取，使得各个序列长度相近，减少浪费

seq2seq

- 定义模型
  - 超参数和占位符

```
encoder_embedding_size, decoder_embedding_size = 30, 30
encoder_hidden_units, decoder_hidden_units = 50, 50
encoder_lstm_layers, decoder_lstm_layers = 2, 2

# [B, T]
encoder_inputs = tf.placeholder(shape=[None, None], dtype=tf.int32, name='encoder_inputs')
decoder_targets = tf.placeholder(shape=[None, None], dtype=tf.int32, name='decoder_targets')
decoder_inputs = tf.placeholder(shape=[None, None], dtype=tf.int32, name='decoder_inputs')
encoder_length = tf.placeholder(shape=[None], dtype=tf.int32, name='encoder_length')
decoder_length = tf.placeholder(shape=[None], dtype=tf.int32, name='decoder_length')
```

seq2seq

- 定义模型
  - 词向量

```
encoder_embedding_matrix = tf.Variable(tf.truncated_normal([vocab_size, encoder_embedding_size],
                                                           mean=0.0, stddev=0.1),
                                       dtype=tf.float32, name="encoder_embedding_matrix")

decoder_embedding_matrix = tf.Variable(tf.truncated_normal([vocab_size, decoder_embedding_size],
                                                           mean=0.0, stddev=0.1),
                                       dtype=tf.float32, name="decoder_embedding_matrix")

# [B, T, D]
encoder_inputs_embedded = tf.nn.embedding_lookup(encoder_embedding_matrix, encoder_inputs)
decoder_inputs_embedded = tf.nn.embedding_lookup(decoder_embedding_matrix, decoder_inputs)
```

seq2seq

- 定义模型
  - 编码器部分：忽略编码器的输出

```
with tf.variable_scope("encoder"):
    encoder_layers = [tf.contrib.rnn.BasicLSTMCell(encoder_hidden_units)
                      for _ in range(encoder_lstm_layers)]
    encoder = tf.contrib.rnn.MultiRNNCell(encoder_layers)

    _, encoder_final_state = tf.nn.dynamic_rnn(
        encoder, encoder_inputs_embedded,
        sequence_length=encoder_length,
        dtype=tf.float32, time_major=False, scope="seq2seq_encoder")
    print(encoder_final_state)
```

seq2seq

- 定义模型
  - 解码器部分：使用 TrainingHelper 和 BasicDecoder

```
with tf.variable_scope("decoder"):
    decoder_layers = [tf.contrib.rnn.BasicLSTMCell(encoder_hidden_units)
                      for _ in range(decoder_lstm_layers)]
    decoder = tf.contrib.rnn.MultiRNNCell(decoder_layers)
    fc_layer = tf.layers.Dense(vocab_size)

    training_helper = tf.contrib.seq2seq.TrainingHelper(decoder_inputs_embedded,
                                                        decoder_length)
    training_decoder = tf.contrib.seq2seq.BasicDecoder(decoder, training_helper,
                                                       encoder_final_state, fc_layer)
```

seq2seq

- 定义模型
  - 解码器存在多种解码方式
  - 训练时采用 teacher forcing
    - 永远把 ground truth 输入给模型，不管模型前一步预测结果是否正确
  - 推断时 ground truth 未知，用别的方式解码
    - greedy decoding：每一次把模型认为概率最大的 token 输入给下一时间步
    - 或者随机采样：每一步从模型预测的概率分布里随机采一个 token 输入给下一时间步
    - 或者 beam search decoding：每次保留 top k 的预测结果，解码得到（近似） k best 序列
    - 还有其他……
    - Helper 和 Decoder 互相协作，定义解码方式

seq2seq

- 定义模型
  - 解码器：得到每一步的 logits

```
logits, final_state, final_sequence_lengths = \
    tf.contrib.seq2seq.dynamic_decode(training_decoder)

# decoder_logits: [B, T, V]
decoder_logits = logits.rnn_output
print("logits: ", decoder_logits)
```

seq2seq

- 定义模型
  - 解码器：loss 是平均交叉熵（用掩码盖住无效部位）

```
# [B, T]
stepwise_cross_entropy = tf.nn.softmax_cross_entropy_with_logits_v2(
    labels=tf.one_hot(decoder_targets, depth=vocab_size, dtype=tf.float32),
    logits=decoder_logits)
print(stepwise_cross_entropy)

mask = tf.sequence_mask(decoder_length,
                        maxlen=tf.reduce_max(decoder_length),
                        dtype=tf.float32)

loss = tf.reduce_sum(stepwise_cross_entropy * mask) / tf.reduce_sum(mask)
train_op = tf.train.AdamOptimizer().minimize(loss)
```

seq2seq

- 定义模型
  - 解码器：推断算法用贪心解码

```
num_sequences_to_decode = tf.placeholder(shape=(), dtype=tf.int32, name="num_seq")
start_tokens = tf.tile([GO], [num_sequences_to_decode])
inference_helper = tf.contrib.seq2seq.GreedyEmbeddingHelper(
    decoder_embedding_matrix, start_tokens, end_token=EOS)

greedy_decoder = tf.contrib.seq2seq.BasicDecoder(
    cell=decoder, helper=inference_helper,
    initial_state=encoder_final_state, output_layer=fc_layer)

greedy_decoding_result, _1, _2 = tf.contrib.seq2seq.dynamic_decode(
    decoder=greedy_decoder, output_time_major=False,
    impute_finished=True, maximum_iterations=20)
```

seq2seq

```
vocab_size = 10 + 1 # 1~10 + 0
max_len = 24
batch_size = 64
PAD = 0
EOS = 0
GO = 0
```

- 得到解码器的输入和输出
  - 互相错位一个时间步
  - 输入要在前面补 <GO> 符号
  - 输出最后有 <EOS>（这里假定 ids 中已经包含了末尾的 EOS）
  - 这个程序里 <PAD>/<EOS>/<GO> 用了同一个 ID，也有的程序会做区分

```
def get_decoder_input_and_output(ids):
    B, T = ids.shape
    go_ids = np.c_[np.zeros([B, 1], dtype=np.int32) + GO, ids]
    return go_ids[:, :-1], go_ids[:, 1:]
```

seq2seq

- 训练过程
  - 喂数据，反复执行 train_op

```
y_in, y_out = get_decoder_input_and_output(y)
# print(x, y, lx, ly, y_in, y_out)
feed = {encoder_inputs: x,
        decoder_inputs: y_in,
        decoder_targets: y_out,
        encoder_length: lx,
        decoder_length: ly}
_, loss_ = sess.run([train_op, loss], feed_dict=feed)
```

seq2seq

- 训练过程
  - 每 100 步让模型推断一下，看看学习进度
  - 可以同时对多个序列进行 seq2seq 解码，这里同时处理 3 条序列

```
number_samples_to_draw = 3
x, y, lx, ly = generate_data(num_samples=number_samples_to_draw)

print('batch {}'.format(batch_id))
print('  minibatch loss: {}'.format(loss_))
feed = {encoder_inputs: x,
        encoder_length: lx,
        num_sequences_to_decode: number_samples_to_draw}
greedy_prediction = sess.run(greedy_decoding_result,
                             feed_dict=feed)
```

seq2seq

- 训练结果
  - step 0

```
Sample x:
[[ 8  6 10  5  8  9  1  5  6  6 10  7  2  2  5  7 10  8  1]
 [10  7  1  3  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
 [ 3 10  8  9  5  6  7  2 10  2  8 10  0  0  0  0  0  0  0]]
Expected y:
[[ 8  6 10  8  6  6 10  2  2 10  8  0]
 [10  0  0  0  0  0  0  0  0  0  0  0]
 [10  8  6  2 10  2  8 10  0  0  0  0]]
Greedy Decoding result:
[[4 4 4 4 4 6 6 6 6 6 6 8 8 8 8 8 8 8 8 8]
 [4 4 4 4 4 4 4 4 6 6 6 6 6 6 8 8 8 8 8 8]
 [3 4 4 4 4 4 6 6 6 6 6 8 8 8 8 8 8 8 8 8]]
```

seq2seq

- 训练结果
  - step 500

```
Sample x:
[[ 6  1  5  7  2 10  2  4  8 10  1  2  3  5  7  9  7  2]
 [ 6  2  9  6  7  9 10  9  6  2  6  5  4  0  0  0  0  0]
 [ 5  5  7  1  9  4 10  3  1  8  6  0  0  0  0  0  0  0]]
Expected y:
[[ 6  2 10  2  4  8 10  2  2  0]
 [ 6  2  6 10  6  2  6  4  0  0]
 [ 4 10  8  6  0  0  0  0  0  0]]
Greedy Decoding result:
[[ 2  2  2  2  4 10  2  8  8  0]
 [ 6  2  6  2  4  4  6  6  0  0]
 [10  4  6  8  0  0  0  0  0  0]]
```

seq2seq

- 训练结果
  - step 1.5k

```
Sample x:
[[ 7  9  9  3  5  7  1  1 10  8  4  4  8  9  0]
 [ 9  1  6 10  8  7  5  7  2  5  6  0  0  0  0]
 [ 7  7 10  8  7  3  5  6  6  6  5  9  4  6  9]]
Expected y:
[[10  8  4  4  8  0  0  0]
 [ 6 10  8  2  6  0  0  0]
 [10  8  6  6  6  4  6  0]]
Greedy Decoding result:
[[10  8  4  4  8  0  0  0]
 [ 6 10  8  2  6  0  0  0]
 [10  8  6  6  6  4  6  0]]
```

seq2seq

- 进阶
  - 如果把任务改成“删掉奇数，剩下偶数序列再多抄写一遍”
  - 这个模型就很难学到了
- 解决办法
  - 使用 Attention
  - 代码：参考前面给出的官方文档，使用 AttentionWrapper

seq2seq

- 其他学习资料
  - 多读源码！
    - https://github.com/google/seq2seq
    - https://github.com/tensorflow/nmt
    - TensorLayer 的源码也很值得读
  - 经典论文
    - *Ilya Sutskever et al.* ***Sequence to Sequence Learning with Neural Networks****. ArXiv 2014*
    - *Dzmitry Bahdanau, KyungHuyn Cho, and Yoshua Bengio.* ***Neural Machine Translation by Jointly Learning to Translate and Align****. ICLR'15*
    - *Yonghui Wu et al.* ***Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation****. ArXiv 2016*
    - *Rico Sennrich et al.* ***Nematus: a Toolkit for Neural Machine Translation****. ArXiv 2017*