[阿里DIN] 从论文源码学习 之 embedding层如何自动更新

原创 罗西的思考 罗西的思考 10月27日

[阿里DIN] 从论文源码学习 之 embedding层如何自动更新

0x00 摘要

Deep Interest Network（DIN）是阿里妈妈精准定向检索及基础算法团队在2017年6月提出的。其针对电子商务领域（e-commerce industry）的CTR预估，重点在于充分利用/挖掘用户历史行为数据中的信息。

本系列文章解读论文以及源码，顺便梳理一些深度学习相关概念和TensorFlow的实现。

本文通过DIN源码 https://github.com/mouna99/dien 分析，来深入展开看看embedding层如何自动更新。

0x01 DIN源码

1.1 问题

在上文中，我们分析了embedding层的作用，但是留了一个问题尚未解答：

- 如何更新mid_embeddings_var这样的embedding层？

即DIN代码中，如下变量怎么更新：

```
self.uid_embeddings_var = tf.get_variable("uid_embedding_var", [n_uid, EMBEDDING_DIM])
self.mid_embeddings_var = tf.get_variable("mid_embedding_var", [n_mid, EMBEDDING_DIM])
self.cat_embeddings_var = tf.get_variable("cat_embedding_var", [n_cat, EMBEDDING_DIM])
```

因为在DIN中，只有这一处初始化 embeddings 的地方，没有找到迭代更新的代码，这会给初学者带来一些困扰。

1.2 答案

先简要说一下答案，embedding层通过 optimizer 进行更新（自动求导），通过 session.run 进行调用更新。

一般意义的 embedding 大多是神经网络倒数第二层的参数权重，只具有整体意义和相对意义，不具备局部意义和绝对含义，这与 embedding 的产生过程有关，

任何 embedding 一开始都是一个随机数，然后随着优化算法，不断迭代更新，最后网络收敛停止迭代的时候，网络各个层的参数就相对固化，得到隐层权重表（此时就相当于得到了我们想要的 embedding），然后在通过查表可以单独查看每个元素的 embedding。

DIN中对应代码如下：

```
# 优化更新（自动求导）
self.optimizer = tf.train.AdamOptimizer(learning_rate=self.lr).minimize(self.loss)

......

# 通过 session.run 进行调用更新
    def train(self, sess, inps):
        if self.use_negsampling:
            loss, accuracy, aux_loss, _ = sess.run([self.loss, self.accuracy, self.aux_loss, self
                self.uid_batch_ph: inps[0],
```

```
                self.mid_batch_ph: inps[1],
                self.cat_batch_ph: inps[2],
                self.mid_his_batch_ph: inps[3],
                self.cat_his_batch_ph: inps[4],
                self.mask: inps[5],
                self.target_ph: inps[6],
                self.seq_len_ph: inps[7],
                self.lr: inps[8],
                self.noclk_mid_batch_ph: inps[9],
                self.noclk_cat_batch_ph: inps[10],
            })
            return loss, accuracy, aux_loss
        else:
            loss, accuracy, _ = sess.run([self.loss, self.accuracy, self.optimizer], feed_dict={
                self.uid_batch_ph: inps[0],
                self.mid_batch_ph: inps[1],
                self.cat_batch_ph: inps[2],
                self.mid_his_batch_ph: inps[3],
                self.cat_his_batch_ph: inps[4],
                self.mask: inps[5],
                self.target_ph: inps[6],
                self.seq_len_ph: inps[7],
                self.lr: inps[8],
            })
            return loss, accuracy, 0
```

这涉及的部分很多，我们需要一一阐释。

0x02 原理

大多数机器学习（深度学习）任务就是最小化损失，在损失函数定义好的情况下，使用一种优化器进行求解最小损失。

而为了让loss下降，深度学习框架常见的优化方式一般采用的是梯度下降（Gradient Descent）算法，这要求对loss公式上的每个op都需要求偏导，然后使用链式法则结合起来。

2.1 随机梯度下降SGD

给定一个可微函数，理论上可以用解析法找到它的最小值：函数的最小值是导数为 0 的点，因此你只需找到所有导数为 0 的点，然后计算函数在其中哪个点具有最小值。

将这一方法应用于神经网络，就是用解析法求出**最小损失函数**对应的所有权重值。可以通过对方程 **gradient(f)(W) = 0** 求解 W 来实现这一方法。

即使用基于梯度的优化方式进行求解，基于当前在随机数据批量上的损失，一点一点地对参数进行调节。由于处理的是一个可微函数，你可以计算出它的梯度，然后沿着梯度的反方向更新权重，损失每次都会变小一点。

1. 抽取训练样本 x 和对应目标y 组成的数据批量。
2. 在 x 上运行网络，得到预测值 y_pred。
3. 计算网络在这批数据上的损失，用于衡量 y_pred 和 y 之间的距离。
4. 计算损失相对于网络参数的梯度［一次反向传播（backward pass）］。
5. 将参数沿着梯度的反方向移动一点，比如 `W -= step * gradient` ，从而使这批数据 上的损失减小一点。

这就叫作**小批量随机梯度下降**（mini-batch stochastic gradient descent，又称为**小批量SGD**）。

术语随机（stochastic）是指每批数据都是随机抽取的（stochastic 是random在科学上的同义词）。

2.2 反向传播

反向传播 算法的训练过程则是根据网络计算得到的 Y_out 和实际的真实结果 Y_label 来计算误差，并且沿着网络反向传播来调整公式中的所有 Wi 和 bi，使**误差**达到最小。强调一下，深度学习里面 BP 的本质目标是让**误差**达到最小，所以要用**误差**对中间出现过的所有影响因素求偏导。

通过反向传播算法优化神经网络是一个迭代的过程。

- 在每次迭代的开始，首先需要选取一小部分训练数据，这一小部分数据叫做一个batch。
- 然后，这个batch 的样例会通过前向传播算法得到神经网络模型的预测结果。因为训练数据都是有正确答案标注的，所以可以计算出当前神经网络模型的预测答案与正确答案之间的差距，计算误差和损失函数。
- 最后，基于这预测值和真实值之间的差距，反向传播算法会相应更新神经网络参数的取值，使得在这个batch 上神经网络模型的预测结果和真实答案更加接近。即首先计算输出层神经元损失函数的梯度，然后计算隐藏层神经元损失函数的梯度。接下来用梯度更新权重。

前向求导是从第一层开始，逐层计算梯度 $\partial / \partial X$ 到最后一层。反向求导是从最后一层开始，逐层计算梯度 $\partial Z / \partial$ 到第一层。前向求导关注的是**输入是怎么影响到每一层**的，反向求导则是关注于**每一层是怎么影响到最终的输出结果**的。

2.3 自动求导

自动求导就是每一个op/layer自己依据自己的输入和输出做前向计算/反向求导，而框架则负责组装调度这些op/layer，表现出来就是你通过框架去定义网络/计算图，框架自动前向计算并自动求导。

常见的深度学习框架里每个op（op指的是最小的计算单元，caffe里叫layer）都预先定义好了forward 和backward（或者叫grad）两个函数，这里的 backward 也就是求导。也就是说每个op的求导都是预先定义好的，或者说是人手推的。

当你定义好了一个神经网络，常见的深度学习框架将其解释为一个dag（有向无环图），dag里每个节点就是op，从loss function这个节点开始，通过链式法则一步一步从后往前计算每一层神经网络的梯度，整个dag梯度计算的最小粒度就是op的 backward 函数（这里是**手动**的），而链式法则则是**自动**的。

TensorFlow也是如此。

TensorFlow 提供的是声明式的编程接口，用户不需要关心求导的细节，只需要定义好模型得到一个loss方程，然后使用TensorFlow实现的各种Optimizer来进行运算即可。

这要求TensorFlow本身提供了每个op的求偏导方法，而且虽然我们使用的是Python的加减乘除运算符，实际上是TensorFlow重载了运算符实际上会创建“Square”这样的op，可以方便用户更容易得构建表达式。

因此TensorFlow的求导，实际上是先提供每一个op求导的数学实现，然后使用链式法则求出整个表达式的导数。

具体我们可以参见RegisterGradient的实现，以及nn_grad.py，math_grad.py等几个文件

这些文件的所有的函数都用RegisterGradient装饰器包装了起来，这些函数都接受两个参数，op和grad。其他的只要注册了op的地方也有各种使用这个装饰器，例如batch。

RegisterGradient使用举例如下：

```
@ops.RegisterGradient("Abs")
def _AbsGrad(op, grad):
  x = op.inputs[0]
  return grad * math_ops.sign(x)
```

RegisterGradient定义如下，就是注册op梯度函数的装饰器：

```
class RegisterGradient(object):
  def __init__(self, op_type):
    if not isinstance(op_type, six.string_types):
      raise TypeError("op_type must be a string")
    self._op_type = op_type

  def __call__(self, f):
    """Registers the function `f` as gradient function for `op_type`."""
    _gradient_registry.register(f, self._op_type)
    return f
```

0x03 优化器

道理说着还不错，但是神经网络是究竟怎么反向传递更新呢？这就需要看Optimizer了。

回到 TensorFlow 的 Python 代码层面，自动求导的部分是靠各种各样的 Optimizer 串起来的：

- 构图的时候只需要写完前向的数据流图部分，TensorFlow 的做法是每一个 Op 在建图的时候就同时包含了它的梯度计算公式，构成前向计算图的时候会自动建立反向部分的计算图，前向计算出来的输入输出会保留下来，留到后向计算的时候用完了才删除。
- 然后在最后加上一个 Optimizer（例如 GradientDescentOptimizer、AdamOptimizer）.
- 最后调用它的 `minimize()` 方法就会自动完成反向部分的数据流图构建。

在DIEN这里，代码如下：

```
ctr_loss = - tf.reduce_mean(tf.log(self.y_hat) * self.target_ph)
self.loss = ctr_loss
if self.use_negsampling:
 self.loss += self.aux_loss
self.optimizer = tf.train.AdamOptimizer(learning_rate=self.lr).minimize(self.loss)
```

3.1 Optimizer基类

TF的optimizer都继承自Optimizer这个类，这个类的方法非常多，几个重要方法是 minimize、compute_gradients、apply_gradients、slot系列。

- **compute_gradients**: 传入loss，如果不传入var_list，那么默认就是所有trainable的variable，返回的是 list of (gradient, variable) pairs。
- **apply_gradients**: 传入 (gradient, variable) pairs，将梯度apply到变量上。具体梯度如何更新到变量，由 _apply_dense、_resource_apply_dense、_apply_sparse、_resource_apply_spars这四个方法实现。
- **minimize**：就是compute_gradients + apply_gradients
- **slot系列**: 输入变量和name，得到的是一个 trainable=False的变量，用来记录optimizer中的中间值，比如在Momentum中，记录momentum。

Optimizer 基类的这个方法为每个实现子类预留了 `_create_slots()` ， `_prepare()` ， `_apply_dense()` ， `_apply_sparse()` 四个接口出来，后面新构建的 Optimizer 只需要重写或者扩展 Optimizer 类的某几个函数即可;

3.2 反向传播过程

整个反向传播过程可分为三步，这三步仅需通过一个minimize()函数完成：

- 逐层计算每一个部分的梯度， `compute_gradients()` ;
- 根据需要对梯度进行处理;
- 把梯度更新到参数上， `apply_gradients();` 即往最小化 loss 的方向更新 var_list 中的每一个参数;

代码如下：

```
  def minimize(self, loss, global_step=None, var_list=None,
               gate_gradients=GATE_OP, aggregation_method=None,
               colocate_gradients_with_ops=False, name=None,
               grad_loss=None):

      grads_and_vars = self.compute_gradients(
          loss, var_list=var_list, gate_gradients=gate_gradients,
          aggregation_method=aggregation_method,
          colocate_gradients_with_ops=colocate_gradients_with_ops,
          grad_loss=grad_loss)

      vars_with_grad = [v for g, v in grads_and_vars if g is not None]

      return self.apply_gradients(grads_and_vars, global_step=global_step, name=name)
```

3.2.1 compute_gradients

该函数用于计算loss对于可训练变量val_list的梯度，最终返回的是元组列表，即 [(gradient, variable),...]。

参数含义:

- **loss**: 需要被优化的Tensor
- **val_list**: Optional list or tuple of `tf.Variable` to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.

基本逻辑如下：

- 根据原本计算图中所有的 op 创建一个顺序的 var_list。即自动找到计算图中所有的 trainable_variables 放到 var_list 里面去，这些就是整个网络中的参数；
- 反向遍历这个list，对每个**需要求导**并且**能够求导**的op（即已经定义好了对应的梯度函数的op）调用其梯度函数；
- 然后沿着原本计算图的方向反向串起另一部分的计算图（输入输出互换，原本的数据 Tensor 换成梯度 Tensor）；即，往图中插入一个 gradients 的 Op，所以反向求导的这个串图的过程就是在这里完成的了；

其中，_get_processor函数可理解为一种快速更新variables的方法，每个processor都会包含一个update_op这样的函数来进行variable更新操作。

变量更新公式：

代码如下：

```
def compute_gradients(self, loss, var_list=None,
                      gate_gradients=GATE_OP,
                      aggregation_method=None,
                      colocate_gradients_with_ops=False,
                      grad_loss=None):

    self._assert_valid_dtypes([loss])
    if grad_loss is not None:
      self._assert_valid_dtypes([grad_loss])
    if var_list is None:
      var_list = (
          variables.trainable_variables() +
```

```
        ops.get_collection(ops.GraphKeys.TRAINABLE_RESOURCE_VARIABLES))
  else:
    var_list = nest.flatten(var_list)

  var_list += ops.get_collection(ops.GraphKeys._STREAMING_MODEL_PORTS)
  processors = [_get_processor(v) for v in var_list]
  var_refs = [p.target() for p in processors]

  grads = gradients.gradients(
      loss, var_refs, grad_ys=grad_loss,
      gate_gradients=(gate_gradients == Optimizer.GATE_OP),
      aggregation_method=aggregation_method,
      colocate_gradients_with_ops=colocate_gradients_with_ops)

  if gate_gradients == Optimizer.GATE_GRAPH:
    grads = control_flow_ops.tuple(grads)

  grads_and_vars = list(zip(grads, var_list))

  return grads_and_vars
```

3.2.2 gradients

gradients 的实际定义在 `tensorflow/python/ops/gradients_impl.py` 中。把整个求导过程抽象成一个 ys=f(xs) 的函数。

简单说，它就是为了计算一组输出张量 `ys = [y0, y1, ...]` 对输入张量 `xs = [x0, x1, ...]` 的梯度，对每个 `xi` 有 `grad_i = sum[dy_j/dx_i for y_j in ys]` 。默认情况下， `grad_loss` 是 `None` ，此时 `grad_ys` 被初始化为全1向量。

gradients 部分参数如下：

- xs 就是 var_list 里面输入的变量列表（在这个过程中其实这里存的是每个变量对应过来在计算图中的 op）。
- 参数中的ys是loss，是是计算损失值的张量，也就是用户业务逻辑最后生成的Tensor的最终节点，从这个节点反推，可以导出全部Graph。
- `grad_ys` 存储计算出的梯度；
- `gate_gradients` 是一个布尔变量，指示所有梯度是否在使用前被算出，如果设为 `True` ，可以避免竞争条件；

这个方法会维护两个重要变量

- 一个队列 `queue` ，队列里存放计算图里所有出度为0的操作符
- 一个字典 `grads` ，字典的键是操作符本身，值是该操作符每个输出端收到的梯度列表

反向传播求梯度时，每从队列中弹出一个操作符，都会把它输出变量的梯度加起来（对应全微分定理）得到 `out_grads` ，然后获取对应的梯度计算函数 `grad_fn` 。操作符 `op` 本身和 `out_grads` 会传递给 `grad_fn` 做参数，求出输入的梯度。

基本逻辑如下：

- 根据原本计算图中所有的 op 创建一个顺序的 list，这个顺序在图上来说其实也是拓扑序；
- 反向遍历这个 list，对每个**需要求导**并且**能够求导**的 op（即已经定义好了对应的梯度函数的 op）调用其梯度函数；
- 然后沿着原本图的方向反向串起另一部分的计算图即可（输入输出互换，原本的数据 Tensor 换成梯度 Tensor）；

具体代码如下：

```
def gradients(ys,
              xs,
              grad_ys=None,
              name="gradients",
              colocate_gradients_with_ops=False,
              gate_gradients=False,
              aggregation_method=None,
              stop_gradients=None):

    to_ops = [t.op for t in ys]
    from_ops = [t.op for t in xs]

    grads = {}
    # Add the initial gradients for the ys.
    for y, grad_y in zip(ys, grad_ys):
      _SetGrad(grads, y, grad_y)

    # Initialize queue with to_ops.
    queue = collections.deque()
    # Add the ops in 'to_ops' into the queue.
    to_ops_set = set()

    for op in to_ops:
      ready = (pending_count[op._id] == 0)
      if ready and op._id not in to_ops_set:
        to_ops_set.add(op._id)
        queue.append(op)

    while queue:
```

```
# generate gradient subgraph for op.
op = queue.popleft()
with _maybe_colocate_with(op, colocate_gradients_with_ops):
  if loop_state:
    loop_state.EnterGradWhileContext(op, before=True)
  out_grads = _AggregatedGrads(grads, op, loop_state, aggregation_method)
  if loop_state:
    loop_state.ExitGradWhileContext(op, before=True)

  if has_out_grads and (op._id not in stop_ops):
    if is_func_call:
      func_call = ops.get_default_graph()._get_function(op.type)
      grad_fn = func_call.python_grad_func
    else:
      try:
        grad_fn = ops.get_gradient_function(op)

  for i, (t_in, in_grad) in enumerate(zip(op.inputs, in_grads)):
    if in_grad is not None:
      if (isinstance(in_grad, ops.Tensor) and
          t_in.dtype != dtypes.resource):
        try:
          in_grad.set_shape(t_in.get_shape())
      _SetGrad(grads, t_in, in_grad)

  if loop_state:
    loop_state.ExitGradWhileContext(op, before=False)
```

3.2.3 apply_gradients

该函数的作用是将 `compute_gradients()` 返回的值作为输入参数对variable进行更新，即根据前面求得的梯度，把梯度进行方向传播给weights和biases进行参数更新。

那为什么 `minimize()` 会分开两个步骤呢？原因是因为在某些情况下我们需要对梯度做一定的修正，例如为了防止梯度消失(gradient vanishing)或者梯度爆炸(gradient explosion)，我们需要事先干预一下以免程序出现**Nan**的尴尬情况；有的时候也许我们需要给计算得到的梯度乘以一个权重或者其他乱七八糟的原因，所以才分开了两个步骤。

基本逻辑如下：

- 对于g, v, p（grads, vars, processors），把它们整合在 tuple(converted_grads_and_vars)；
- 遍历参数列表 v，对于每一个参数应用 self._create_slots函数，以创建一些优化器自带的一些参数;

- 调用 prepare()函数，在apply梯度前创建好所有必须的tensors;
- 遍历 `grad, var, processor in converted_grads_and_vars` ，应用ops.colocate_with(var)，作用是保证每个参数var的更新都在同一个device上;
- ops.control_dependencies()函数用来控制计算流图的，给图中的某些节点指定计算的顺序;
- 对每个 variable 本身应用 assign，体现在 `update_ops.append(processor.update_op(self, grad))` ，如果有 `global_step` 的话，global_step需加个1。
- 最后将返回一个 `train_op` 。 `train_op` 是通常训练过程中，client为session的fetches提供的参数之一，也就是这个Operation被执行之后，模型的参数将会完成更新，并开始下一个batch的训练。那么这也就意味着，这个方法中涉及到的计算图将会实现说明文档中的训练逻辑。

具体代码是：

```
def apply_gradients(self, grads_and_vars, global_step=None, name=None):

    grads_and_vars = tuple(grads_and_vars)  # Make sure repeat iteration works.

    converted_grads_and_vars = []
    for g, v in grads_and_vars:
      if g is not None:
        # Convert the grad to Tensor or IndexedSlices if necessary.
        g = ops.convert_to_tensor_or_indexed_slices(g)

      p = _get_processor(v)
      converted_grads_and_vars.append((g, v, p))

    converted_grads_and_vars = tuple(converted_grads_and_vars)
    var_list = [v for g, v, _ in converted_grads_and_vars if g is not None]

    with ops.control_dependencies(None):
      self._create_slots([_get_variable_for(v) for v in var_list])
    update_ops = []
    with ops.name_scope(name, self._name) as name:
      self._prepare()
      for grad, var, processor in converted_grads_and_vars:
        if grad is None:
          continue
        scope_name = var.op.name if context.in_graph_mode() else ""
        with ops.name_scope("update_" + scope_name), ops.colocate_with(var):
          update_ops.append(processor.update_op(self, grad))
      if global_step is None:
        apply_updates = self._finish(update_ops, name)
      else:
```

```
        with ops.control_dependencies([self._finish(update_ops, "update")]):
          with ops.colocate_with(global_step):
            apply_updates = state_ops.assign_add(global_step, 1, name=name).op

      train_op = ops.get_collection_ref(ops.GraphKeys.TRAIN_OP)
      if apply_updates not in train_op:
        train_op.append(apply_updates)

      return apply_updates
```

3.3 AdamOptimizer

DIEN使用的是AdamOptimizer优化器。

Adam 这个名字来源于**自适应矩估计（Adaptive Moment Estimation）**，也是梯度下降算法的一种变形，但是每次迭代参数的学习率都有一定的范围，不会因为梯度很大而导致学习率（步长）也变得很大，参数的值相对比较稳定。

```
概率论中矩的含义是：如果一个随机变量 X 服从某个分布，X 的一阶矩是 E(X)，也就是样本平均值，X 的二阶矩就是 E
```

Adam 算法利用梯度的一阶矩估计和二阶矩估计动态调整每个参数的学习率。TensorFlow提供的tf.train.AdamOptimizer可控制学习速度，经过偏置校正后，每一次迭代学习率都有个确定范围，使得参数比较平稳。

在利用计算好的导数对权重进行修正时，对**Embedding矩阵的梯度进行特殊处理，只更新局部**，见optimization.py中Adagrad.update函数。

3.3.1 _prepare

在 `_prepare` 函数中通过 `convert_to_tensor` 方法来存储了输入参数的 Tensor 版本。

```
  def _prepare(self):
    self._lr_t = ops.convert_to_tensor(self._lr, name="learning_rate")
    self._beta1_t = ops.convert_to_tensor(self._beta1, name="beta1")
    self._beta2_t = ops.convert_to_tensor(self._beta2, name="beta2")
    self._epsilon_t = ops.convert_to_tensor(self._epsilon, name="epsilon")
```

3.3.2 _create_slots

_create_slots 函数用来创建参数，比如 _beta1_power，_beta2_power

```
def _create_slots(self, var_list):

  first_var = min(var_list, key=lambda x: x.name)

  create_new = self._beta1_power is None
  if not create_new and context.in_graph_mode():
    create_new = (self._beta1_power.graph is not first_var.graph)

  if create_new:
    with ops.colocate_with(first_var):
      self._beta1_power = variable_scope.variable(self._beta1,
                                                  name="beta1_power",
                                                  trainable=False)
      self._beta2_power = variable_scope.variable(self._beta2,
                                                  name="beta2_power",
                                                  trainable=False)
  # Create slots for the first and second moments.
  for v in var_list:
    self._zeros_slot(v, "m", self._name)
    self._zeros_slot(v, "v", self._name)
```

函数** _apply_dense 和 _resource_apply_dense 的实现中分别使用了 training_ops.apply_adam 和 training_ops.resource_apply_adam **方法。

函数** _apply_sparse 和 _resource_apply_sparse 主要用在稀疏向量的更新操作上，而具体的实现是在函数 _apply_sparse_shared **中。

_apply_sparse_shared 函数，首先获取所需要的参数值并存储到变量里，接着按照 Adam 算法的流程，首先计算学习率

，接着计算两个 Momentum，由于是稀疏 tensor 的更新，所以在算出更新值之后要使用** scatter_add 来完成加法操作， 最后将 var_update 和 m_t 、 v_t 的更新操作放进 control_flow_ops.group **中。

0x04 Session.run

优化器已经搭建好，剩下就是调用 session.run 进行更新。

调用一次 run 是执行一遍数据流图， 在 TensorFlow 的训练代码中通常是在一个循环中多次调用 `sess.run()` ，一次 run 即为训练过程中的一步。

fetches 是 run 方法的一个输入参数，这个参数可以是很多种形式的数据，run 最后的 返回值也会和 fetches 有相同的结构。

至此，DIN分析暂时告一段落，下篇开始 DIEN 的分析，敬请期待。

0xFF 参考

TensorFlow SyncReplicasOptimizer 解读

tensorflow中有向图（计算图、Graph）、上下文环境（Session）和执行流程

TensorFlow 拆包（一）：Session.Run ()

TensorFlow 源码大坑(2) Session

tensorflow源码分析（五）session.run()

Tensorflow中优化器--AdamOptimizer详解

【TensorFlow】优化器AdamOptimizer的源码分析

图解tensorflow源码

TensorFlow中Session、Graph、Operation以及Tensor详解

TensorFlow 拆包（二）：TF 的数据流模型实现以及自动求导

分布式Tensorflow中同步梯度更新tf.train.SyncReplicasOptimizer解读（backup_worker的用法）

TensorFlow实战系列3--实现反向传播

Optimizer in Tensorflow

tensorflow optimizer源码阅读笔记

TensorFlow学习笔记之--[compute_gradients和apply_gradients原理浅析]

tensorflow学习（三）

Tensorflow 是如何求导的？

道理我都懂，但是神经网络反向传播时的梯度到底怎么求？