tf.contrib.opt.LazyAdamOptimizer

Contents

Class `LazyAdamOptimizer`

Inherits From: `AdamOptimizer`

Defined in `tensorflow/contrib/opt/python/training/lazy_adam_optimizer.py`.

Variant of the Adam optimizer that handles sparse updates more efficiently.

The original Adam algorithm maintains two moving-average accumulators for each trainable variable; the accumulators are updated at every step. This class provides lazier handling of gradient updates for sparse variables. It only updates moving-average accumulators for sparse variable indices that appear in the current batch, rather than updating the accumulators for all indices. Compared with the original Adam optimizer, it can provide large improvements in model training throughput for some applications. However, it provides slightly different semantics than the original Adam algorithm, and may lead to different empirical results.

Methods

`__init__`

```
__init__(
    learning_rate=0.001,
    beta1=0.9,
    beta2=0.999,
    epsilon=1e-08,
    use_locking=False,
    name='Adam'
)
```

Construct a new Adam optimizer.

Initialization:

```
m_0 <- 0 (Initialize initial 1st moment vector)
v_0 <- 0 (Initialize initial 2nd moment vector)
t <- 0 (Initialize timestep)
```

The update rule for `variable` with gradient `g` uses an optimization described at the end of section2 of the paper:

```
t <- t + 1
lr_t <- learning_rate * sqrt(1 - beta2^t) / (1 - beta1^t)

m_t <- beta1 * m_{t-1} + (1 - beta1) * g
v_t <- beta2 * v_{t-1} + (1 - beta2) * g * g
variable <- variable - lr_t * m_t / (sqrt(v_t) + epsilon)
```

The default value of 1e-8 for epsilon might not be a good default in general. For example, when training an Inception network on ImageNet a current good choice is 1.0 or 0.1. Note that since AdamOptimizer uses the formulation just before Section 2.1 of the Kingma and Ba paper rather than the formulation in Algorithm 1, the "epsilon" referred to here is "epsilon hat" in the paper.

The sparse implementation of this algorithm (used when the gradient is an IndexedSlices object, typically because of `tf.gather` or an embedding lookup in the forward pass) does apply momentum to variable slices even if they were not used in the forward pass (meaning they have a gradient equal to zero). Momentum decay (beta1) is also applied to the entire momentum accumulator. This means that the sparse behavior is equivalent to the dense behavior (in contrast to some momentum implementations which ignore momentum unless a variable slice was actually used).

Args:

- `learning_rate` : A Tensor or a floating point value. The learning rate.
- `beta1` : A float value or a constant float tensor. The exponential decay rate for the 1st moment estimates.
- `beta2` : A float value or a constant float tensor. The exponential decay rate for the 2nd moment estimates.
- `epsilon` : A small constant for numerical stability. This epsilon is "epsilon hat" in the Kingma and Ba paper (in the formula just before Section 2.1), not the epsilon in Algorithm 1 of the paper.
- `use_locking` : If True use locks for update operations.
- `name` : Optional name for the operations created when applying gradients. Defaults to "Adam".

`apply_gradients`

```
apply_gradients(
    grads_and_vars,
    global_step=None,
    name=None
)
```

Apply gradients to variables.

This is the second part of `minimize()` . It returns an `Operation` that applies gradients.

Args:

- `grads_and_vars` : List of (gradient, variable) pairs as returned by `compute_gradients()` .
- `global_step` : Optional `Variable` to increment by one after the variables have been updated.
- `name` : Optional name for the returned operation. Default to the name passed to the `Optimizer` constructor.

Returns:

An `Operation` that applies the specified gradients. If `global_step` was not None, that operation also increments `global_step` .

Raises:

- `TypeError`: If `grads_and_vars` is malformed.
- `ValueError`: If none of the variables have gradients.

`compute_gradients`

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

Compute gradients of `loss` for the variables in `var_list`.

This is the first part of `minimize()`. It returns a list of (gradient, variable) pairs where "gradient" is the gradient for "variable". Note that "gradient" can be a `Tensor`, an `IndexedSlices`, or `None` if there is no gradient for the given variable.

Args:

- `loss`: A Tensor containing the value to minimize.
- `var_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 `GraphKey.TRAINABLE_VARIABLES`.
- `gate_gradients`: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or `GATE_GRAPH`.
- `aggregation_method`: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.
- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.
- `grad_loss`: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

A list of (gradient, variable) pairs. Variable is always present, but gradient can be `None`.

Raises:

- `TypeError`: If `var_list` contains anything else than `Variable` objects.
- `ValueError`: If some arguments are invalid.

`get_name`

```
get_name()
```

`get_slot`

```
get_slot(
    var,
    name
)
```

Return a slot named `name` created for `var` by the Optimizer.

Some `Optimizer` subclasses use additional variables. For example `Momentum` and `Adagrad` use variables to accumulate updates. This method gives access to these `Variable` objects if for some reason you need them.

Use `get_slot_names()` to get the list of slot names created by the `Optimizer`.

Args:

- `var`: A variable passed to `minimize()` or `apply_gradients()`.
- `name`: A string.

Returns:

The `Variable` for the slot if it was created, `None` otherwise.

`get_slot_names`

```
get_slot_names()
```

Return a list of the names of slots created by the `Optimizer`.

See `get_slot()`.

Returns:

A list of strings.

`minimize`

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

Add operations to minimize `loss` by updating `var_list`.

This method simply combines calls `compute_gradients()` and `apply_gradients()`. If you want to process the gradient before applying them call `compute_gradients()` and `apply_gradients()` explicitly instead of using this function.

Args:

- `loss`: A `Tensor` containing the value to minimize.
- `global_step`: Optional `Variable` to increment by one after the variables have been updated.
- `var_list`: Optional list or tuple of `Variable` objects to update to minimize `loss`. Defaults to the list of variables collected in the graph under the key `GraphKeys.TRAINABLE_VARIABLES`.
- `gate_gradients`: How to gate the computation of gradients. Can be `GATE_NONE`, `GATE_OP`, or `GATE_GRAPH`.
- `aggregation_method`: Specifies the method used to combine gradient terms. Valid values are defined in the class `AggregationMethod`.

- `colocate_gradients_with_ops`: If True, try colocating gradients with the corresponding op.
- `name`: Optional name for the returned operation.
- `grad_loss`: Optional. A `Tensor` holding the gradient computed for `loss`.

Returns:

An Operation that updates the variables in `var_list`. If `global_step` was not `None`, that operation also increments `global_step`.

Raises:

- `ValueError`: If some of the variables are not `Variable` objects.

Class Members

GATE_GRAPH

GATE_NONE

GATE_OP