tf.train.ExponentialMovingAverage

Contents

Class **`ExponentialMovingAverage`**

Defined in `tensorflow/python/training/moving_averages.py`.

See the guide: Training > Moving Averages

Maintains moving averages of variables by employing an exponential decay.

When training a model, it is often beneficial to maintain moving averages of the trained parameters. Evaluations that use averaged parameters sometimes produce significantly better results than the final trained values.

The `apply()` method adds shadow copies of trained variables and add ops that maintain a moving average of the trained variables in their shadow copies. It is used when building the training model. The ops that maintain moving averages are typically run after each training step. The `average()` and `average_name()` methods give access to the shadow variables and their names. They are useful when building an evaluation model, or when restoring a model from a checkpoint file. They help use the moving averages in place of the last trained values for evaluations.

The moving averages are computed using exponential decay. You specify the decay value when creating the `ExponentialMovingAverage` object. The shadow variables are initialized with the same initial values as the trained variables. When you run the ops to maintain the moving averages, each shadow variable is updated with the formula:

`shadow_variable -= (1 - decay) * (shadow_variable - variable)`

This is mathematically equivalent to the classic formula below, but the use of an `assign_sub` op (the `"-="` in the formula) allows concurrent lockless updates to the variables:

`shadow_variable = decay * shadow_variable + (1 - decay) * variable`

Reasonable values for `decay` are close to 1.0, typically in the multiple-nines range: 0.999, 0.9999, etc.

Example usage when creating a training model:

```
# Create variables.
var0 = tf.Variable(...)
var1 = tf.Variable(...)
# ... use the variables to build a training model...
...
# Create an op that applies the optimizer.  This is what we usually
# would use as a training op.
opt_op = opt.minimize(my_loss, [var0, var1])

# Create an ExponentialMovingAverage object
ema = tf.train.ExponentialMovingAverage(decay=0.9999)

with tf.control_dependencies([opt_op]):
    # Create the shadow variables, and add ops to maintain moving averages
    # of var0 and var1. This also creates an op that will update the moving
    # averages after each training step.  This is what we will use in place
    # of the usual training op.
    training_op = ema.apply([var0, var1])

...train the model by running training_op...
```

There are two ways to use the moving averages for evaluations:

- Build a model that uses the shadow variables instead of the variables. For this, use the `average()` method which returns the shadow variable for a given variable.
- Build a model normally but load the checkpoint files to evaluate by using the shadow variable names. For this use the `average_name()` method. See the `tf.train.Saver` for more information on restoring saved variables.

Example of restoring the shadow variable values:

```
# Create a Saver that loads variables from their saved shadow values.
shadow_var0_name = ema.average_name(var0)
shadow_var1_name = ema.average_name(var1)
saver = tf.train.Saver({shadow_var0_name: var0, shadow_var1_name: var1})
saver.restore(...checkpoint filename...)
# var0 and var1 now hold the moving average values
```

Methods

`__init__`

```
__init__(
    decay,
    num_updates=None,
    zero_debias=False,
    name='ExponentialMovingAverage'
)
```

Creates a new ExponentialMovingAverage object.

The `apply()` method has to be called to create shadow variables and add ops to maintain moving averages.

The optional `num_updates` parameter allows one to tweak the decay rate dynamically. It is typical to pass the count of training steps, usually kept in a variable that is incremented at each step, in which case the decay rate is lower at the start of training. This makes moving averages move faster. If passed, the actual decay rate used is:

`min(decay, (1 + num_updates) / (10 + num_updates))`

Args:

- `decay`: Float. The decay to use.
- `num_updates`: Optional count of number of updates applied to variables.
- `zero_debias`: If `True`, zero debias moving-averages that are initialized with tensors.
- `name`: String. Optional prefix name to use for the name of ops added in `apply()`.

`apply`

```
apply(var_list=None)
```

Maintains moving averages of variables.

`var_list` must be a list of `Variable` or `Tensor` objects. This method creates shadow variables for all elements of `var_list`. Shadow variables for `Variable` objects are initialized to the variable's initial value. They will be added to the `GraphKeys.MOVING_AVERAGE_VARIABLES` collection. For `Tensor` objects, the shadow variables are initialized to 0 and zero debiased (see docstring in `assign_moving_average` for more details).

shadow variables are created with `trainable=False` and added to the `GraphKeys.ALL_VARIABLES` collection. They will be returned by calls to `tf.global_variables()`.

Returns an op that updates all shadow variables as described above.

Note that `apply()` can be called multiple times with different lists of variables.

Args:

- `var_list`: A list of Variable or Tensor objects. The variables and Tensors must be of types float16, float32, or float64.

Returns:

An Operation that updates the moving averages.

Raises:

- `TypeError`: If the arguments are not all float16, float32, or float64.
- `ValueError`: If the moving average of one of the variables is already being computed.

`average`

```
average(var)
```

Returns the `Variable` holding the average of `var`.

Args:

- `var`: A `Variable` object.

Returns:

A `Variable` object or `None` if the moving average of `var` is not maintained.

`average_name`

```
average_name(var)
```

Returns the name of the `Variable` holding the average for `var`.

The typical scenario for `ExponentialMovingAverage` is to compute moving averages of variables during training, and restore the variables from the computed moving averages during evaluations.

To restore variables, you have to know the name of the shadow variables. That name and the original variable can then be passed to a `Saver()` object to restore the variable from the moving average value with: `saver = tf.train.Saver({ema.average_name(var): var})`

`average_name()` can be called whether or not `apply()` has been called.

Args:

- `var`: A `Variable` object.

Returns:

A string: The name of the variable that will be used or was used by the `ExponentialMovingAverage class` to hold the moving average of `var`.

`variables_to_restore`

```
variables_to_restore(moving_avg_variables=None)
```

Returns a map of names to `Variables` to restore.

If a variable has a moving average, use the moving average variable name as the restore name; otherwise, use the variable name.

For example,

```
  variables_to_restore = ema.variables_to_restore()
  saver = tf.train.Saver(variables_to_restore)
```

Below is an example of such mapping:

```
  conv/batchnorm/gamma/ExponentialMovingAverage: conv/batchnorm/gamma,
  conv_4/conv2d_params/ExponentialMovingAverage: conv_4/conv2d_params,
  global_step: global_step
```

Args:

- `moving_avg_variables`: a list of variables that require to use of the moving variable name to be restored. If None, it will default to variables.moving_average_variables() + variables.trainable_variables()

Returns:

A map from restore_names to variables. The restore_name can be the moving_average version of the variable name if it exist, or the original variable name.