Optimizers

Usage with `compile()` & `fit()`

An optimizer is one of the two arguments required for compiling a Keras model:

```
from tensorflow import keras
from tensorflow.keras import layers

model = keras.Sequential()
model.add(layers.Dense(64, kernel_initializer='uniform', input_shape=(10,)))
model.add(layers.Activation('softmax'))

opt = keras.optimizers.Adam(learning_rate=0.01)
model.compile(loss='categorical_crossentropy', optimizer=opt)
```

You can either instantiate an optimizer before passing it to `model.compile()` , as in the above example, or you can pass it by its string identifier. In the latter case, the default parameters for the optimizer will be used.

```
# pass optimizer by name: default parameters will be used
model.compile(loss='categorical_crossentropy', optimizer='adam')
```

Usage in a custom training loop

When writing a custom training loop, you would retrieve gradients via a `tf.GradientTape` instance, then call `optimizer.apply_gradients()` to update your weights:

```
# Instantiate an optimizer.
optimizer = tf.keras.optimizers.Adam()

# Iterate over the batches of a dataset.
for x, y in dataset:
    # Open a GradientTape.
    with tf.GradientTape() as tape:
        # Forward pass.
        logits = model(x)
        # Loss value for this batch.
        loss_value = loss_fn(y, logits)

    # Get gradients of loss wrt the weights.
    gradients = tape.gradient(loss_value, model.trainable_weights)

    # Update the weights of the model.
    optimizer.apply_gradients(zip(gradients, model.trainable_weights))
```

Note that when you use `apply_gradients`, the optimizer does not apply gradient clipping to the gradients: if you want gradient clipping, you would have to do it by hand before calling the method.

Learning rate decay / scheduling

You can use a learning rate schedule to modulate how the learning rate of your optimizer changes over time:

```
lr_schedule = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1e-2,
    decay_steps=10000,
    decay_rate=0.9)
optimizer = keras.optimizers.SGD(learning_rate=lr_schedule)
```

Check out the learning rate schedule API documentation for a list of available schedules.

Available optimizers

- SGD
- RMSprop
- Adam
- Adadelta
- Adagrad
- Adamax
- Nadam
- Ftrl

Core Optimizer API

These methods and attributes are common to all Keras optimizers.

`apply_gradients` method

```
Optimizer.apply_gradients(
    grads_and_vars, name=None, experimental_aggregate_gradients=True
)
```

Apply gradients to variables.

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

The method sums gradients from all replicas in the presence of `tf.distribute.Strategy` by default. You can aggregate gradients yourself by passing `experimental_aggregate_gradients=False`.

Example

```
grads = tape.gradient(loss, vars)
grads = tf.distribute.get_replica_context().all_reduce('sum', grads)
# Processing aggregated gradients.
optimizer.apply_gradients(zip(grads, vars),
    experimental_aggregate_gradients=False)
```

Arguments

- **grads_and_vars**: List of (gradient, variable) pairs.
- **name**: Optional name for the returned operation. Default to the name passed to the `Optimizer` constructor.
- **experimental_aggregate_gradients**: Whether to sum gradients from different replicas in the presence of `tf.distribute.Strategy`. If False, it's user responsibility to aggregate the gradients. Default to True.

Returns

An `Operation` that applies the specified gradients. The `iterations` will be automatically increased by 1.

Raises

- **TypeError**: If `grads_and_vars` is malformed.
- **ValueError**: If none of the variables have gradients.
- **RuntimeError**: If called in a cross-replica context.

`weights` property

```
tf.keras.optimizers.Optimizer.weights
```

Returns variables of this Optimizer based on the order created.

`get_weights` method

```
Optimizer.get_weights()
```

Returns the current weights of the optimizer.

The weights of an optimizer are its state (ie, variables). This function returns the weight values associated with this optimizer as a list of Numpy arrays. The first value is always the iterations count of the optimizer, followed by the optimizer's state variables in the order they were created. The returned list can in turn be used to load state into similarly parameterized optimizers.

For example, the RMSprop optimizer for this simple model returns a list of three values-- the iteration count, followed by the root-mean-square value of the kernel and bias of the single Dense layer:

```
>>> opt = tf.keras.optimizers.RMSprop()
>>> m = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
>>> m.compile(opt, loss='mse')
>>> data = np.arange(100).reshape(5, 20)
>>> labels = np.zeros(5)
>>> print('Training'); results = m.fit(data, labels)
Training ...
>>> len(opt.get_weights())
3
```

Returns

Weights values as a list of numpy arrays.

`set_weights` method

```
Optimizer.set_weights(weights)
```

Set the weights of the optimizer.

The weights of an optimizer are its state (ie, variables). This function takes the weight values associated with this optimizer as a list of Numpy arrays. The first value is always the iterations count of the optimizer, followed by the optimizer's state variables in the order they are created. The passed values are used to set the new state of the optimizer.

For example, the RMSprop optimizer for this simple model takes a list of three values-- the iteration count, followed by the root-mean-square value of the kernel and bias of the single Dense layer:

```
>>> opt = tf.keras.optimizers.RMSprop()
>>> m = tf.keras.models.Sequential([tf.keras.layers.Dense(10)])
>>> m.compile(opt, loss='mse')
>>> data = np.arange(100).reshape(5, 20)
>>> labels = np.zeros(5)
>>> print('Training'); results = m.fit(data, labels)
Training ...
>>> new_weights = [np.array(10), np.ones([20, 10]), np.zeros([10])]
>>> opt.set_weights(new_weights)
>>> opt.iterations
<tf.Variable 'RMSprop/iter:0' shape=() dtype=int64, numpy=10>
```

Arguments

- **weights**: weight values as a list of numpy arrays.