Making new layers and models via subclassing

Author: fchollet
Date created: 2019/03/01
Last modified: 2023/06/25
Description: Complete guide to writing `Layer` and `Model` objects from scratch.

Introduction

This guide will cover everything you need to know to build your own subclassed layers and models. In particular, you'll learn about the following features:

- The `Layer` class
- The `add_weight()` method
- Trainable and non-trainable weights
- The `build()` method
- Making sure your layers can be used with any backend
- The `add_loss()` method
- The `training` argument in `call()`
- The `mask` argument in `call()`
- Making sure your layers can be serialized

Let's dive in.

Setup

```
import numpy as np
import keras
from keras import ops
from keras import layers
```

The `Layer` class: the combination of state (weights) and some computation

One of the central abstractions in Keras is the `Layer` class. A layer encapsulates both a state (the layer's "weights") and a transformation from inputs to outputs (a "call", the layer's forward pass).

Here's a densely-connected layer. It has two state variables: the variables `w` and `b`.

```

class Linear(keras.layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super().__init__()
        self.w = self.add_weight(
            shape=(input_dim, units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(shape=(units,), initializer="zeros", trainable=True)

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b

```

You would use a layer by calling it on some tensor input(s), much like a Python function.

```
x = ops.ones((2, 2))
linear_layer = Linear(4, 2)
y = linear_layer(x)
print(y)
```

Note that the weights `w` and `b` are automatically tracked by the layer upon being set as layer attributes:

```
assert linear_layer.weights == [linear_layer.w, linear_layer.b]
```

Layers can have non-trainable weights

Besides trainable weights, you can add non-trainable weights to a layer as well. Such weights are meant not to be taken into account during backpropagation, when you are training the layer.

Here's how to add and use a non-trainable weight:

```

class ComputeSum(keras.layers.Layer):
    def __init__(self, input_dim):
        super().__init__()
        self.total = self.add_weight(
            initializer="zeros", shape=(input_dim,), trainable=False
        )

    def call(self, inputs):
        self.total.assign_add(ops.sum(inputs, axis=0))
        return self.total


x = ops.ones((2, 2))
my_sum = ComputeSum(2)
y = my_sum(x)
print(y.numpy())
y = my_sum(x)
print(y.numpy())
```

It's part of `layer.weights`, but it gets categorized as a non-trainable weight:

```
print("weights:", len(my_sum.weights))
print("non-trainable weights:", len(my_sum.non_trainable_weights))

# It's not included in the trainable weights:
print("trainable_weights:", my_sum.trainable_weights)
```

Best practice: deferring weight creation until the shape of the inputs is known

Our `Linear` layer above took an `input_dim` argument that was used to compute the shape of the weights `w` and `b` in `__init__()`:

```

class Linear(keras.layers.Layer):
    def __init__(self, units=32, input_dim=32):
        super().__init__()
        self.w = self.add_weight(
            shape=(input_dim, units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(shape=(units,), initializer="zeros", trainable=True)

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b

```

In many cases, you may not know in advance the size of your inputs, and you would like to lazily create weights when that value becomes known, some time after instantiating the layer.

In the Keras API, we recommend creating layer weights in the `build(self, inputs_shape)` method of your layer. Like this:

```
class Linear(keras.layers.Layer):
    def __init__(self, units=32):
        super().__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(
            shape=(self.units,), initializer="random_normal", trainable=True
        )

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b
```

The `__call__()` method of your layer will automatically run build the first time it is called. You now have a layer that's lazy and thus easier to use:

```
# At instantiation, we don't know on what inputs this is going to get called
linear_layer = Linear(32)

# The layer's weights are created dynamically the first time the layer is called
y = linear_layer(x)
```

Implementing `build()` separately as shown above nicely separates creating weights only once from using weights in every call.

Layers are recursively composable

If you assign a Layer instance as an attribute of another Layer, the outer layer will start tracking the weights created by the inner layer.

We recommend creating such sublayers in the `__init__()` method and leave it to the first `__call__()` to trigger building their weights.

```

class MLPBlock(keras.layers.Layer):
    def __init__(self):
        super().__init__()
        self.linear_1 = Linear(32)
        self.linear_2 = Linear(32)
        self.linear_3 = Linear(1)

    def call(self, inputs):
        x = self.linear_1(inputs)
        x = keras.activations.relu(x)
        x = self.linear_2(x)
        x = keras.activations.relu(x)
        return self.linear_3(x)


mlp = MLPBlock()
y = mlp(ops.ones(shape=(3, 64)))  # The first call to the `mlp` will create the weights
print("weights:", len(mlp.weights))
print("trainable weights:", len(mlp.trainable_weights))
```

Backend-agnostic layers and backend-specific layers

As long as a layer only uses APIs from the `keras.ops` namespace (or other Keras namespaces such as `keras.activations`, `keras.random`, or `keras.layers`), then it can be used with any backend -- TensorFlow, JAX, or PyTorch.

All layers you've seen so far in this guide work with all Keras backends.

The `keras.ops` namespace gives you access to:

- The NumPy API, e.g. `ops.matmul`, `ops.sum`, `ops.reshape`, `ops.stack`, etc.
- Neural networks-specific APIs such as `ops.softmax`, `ops` .conv , ops.binary_crossentropy , ops.relu`, etc.

You can also use backend-native APIs in your layers (such as `tf.nn` functions), but if you do this, then your layer will only be usable with the backend in question. For instance, you could write the following JAX-specific layer using `jax.numpy`:

```
import jax


class Linear(keras.layers.Layer):
```

```
    ...

    def call(self, inputs):
        return jax.numpy.matmul(inputs, self.w) + self.b
```

This would be the equivalent TensorFlow-specific layer:

```
import tensorflow as tf

class Linear(keras.layers.Layer):
    ...

    def call(self, inputs):
        return tf.matmul(inputs, self.w) + self.b
```

And this would be the equivalent PyTorch-specific layer:

```
import torch

class Linear(keras.layers.Layer):
    ...

    def call(self, inputs):
        return torch.matmul(inputs, self.w) + self.b
```

Because cross-backend compatibility is a tremendously useful property, we strongly recommend that you seek to always make your layers backend-agnostic by leveraging only Keras APIs.

The `add_loss()` method

When writing the `call()` method of a layer, you can create loss tensors that you will want to use later, when writing your training loop. This is doable by calling `self.add_loss(value)`:

```

# A layer that creates an activity regularization loss
class ActivityRegularizationLayer(keras.layers.Layer):
    def __init__(self, rate=1e-2):
        super().__init__()
        self.rate = rate

    def call(self, inputs):
        self.add_loss(self.rate * ops.mean(inputs))
        return inputs

```

These losses (including those created by any inner layer) can be retrieved via `layer.losses`. This property is reset at the start of every `__call__()` to the top-level layer, so that `layer.losses` always contains the loss values created during the last forward pass.

```

class OuterLayer(keras.layers.Layer):
    def __init__(self):
        super().__init__()
        self.activity_reg = ActivityRegularizationLayer(1e-2)

    def call(self, inputs):
        return self.activity_reg(inputs)


layer = OuterLayer()
assert len(layer.losses) == 0  # No losses yet since the layer has never been called

_ = layer(ops.zeros((1, 1)))
assert len(layer.losses) == 1  # We created one loss value

# `layer.losses` gets reset at the start of each __call__
_ = layer(ops.zeros((1, 1)))
assert len(layer.losses) == 1  # This is the loss created during the call above
```

In addition, the `loss` property also contains regularization losses created for the weights of any inner layer:

```python

class OuterLayerWithKernelRegularizer(keras.layers.Layer):
    def __init__(self):
        super().__init__()
        self.dense = keras.layers.Dense(
            32, kernel_regularizer=keras.regularizers.l2(1e-3)
        )

    def call(self, inputs):
        return self.dense(inputs)


layer = OuterLayerWithKernelRegularizer()
_ = layer(ops.zeros((1, 1)))

# This is `1e-3 * sum(layer.dense.kernel ** 2)`,
# created by the `kernel_regularizer` above.
print(layer.losses)
```

These losses are meant to be taken into account when writing custom training loops.

They also work seamlessly with `fit()` (they get automatically summed and added to the main loss, if any):

```python
inputs = keras.Input(shape=(3,))
outputs = ActivityRegularizationLayer()(inputs)
model = keras.Model(inputs, outputs)

# If there is a loss passed in `compile`, the regularization
# losses get added to it
model.compile(optimizer="adam", loss="mse")
model.fit(np.random.random((2, 3)), np.random.random((2, 3)))

# It's also possible not to pass any loss in `compile`,
# since the model already has a loss to minimize, via the `add_loss`
# call during the forward pass!
model.compile(optimizer="adam")
model.fit(np.random.random((2, 3)), np.random.random((2, 3)))
```

You can optionally enable serialization on your layers

If you need your custom layers to be serializable as part of a [Functional model](), you can optionally implement a `get_config()` method:

```

class Linear(keras.layers.Layer):
    def __init__(self, units=32):
        super().__init__()
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(
            shape=(self.units,), initializer="random_normal", trainable=True
        )

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b

    def get_config(self):
        return {"units": self.units}


# Now you can recreate the layer from its config:
layer = Linear(64)
config = layer.get_config()
print(config)
new_layer = Linear.from_config(config)
```

Note that the `__init__()` method of the base `Layer` class takes some keyword arguments, in particular a `name` and a `dtype`. It's good practice to pass these arguments to the parent class in `__init__()` and to include them in the layer config:

```python

class Linear(keras.layers.Layer):
    def __init__(self, units=32, **kwargs):
        super().__init__(**kwargs)
        self.units = units

    def build(self, input_shape):
        self.w = self.add_weight(
            shape=(input_shape[-1], self.units),
            initializer="random_normal",
            trainable=True,
        )
        self.b = self.add_weight(
            shape=(self.units,), initializer="random_normal", trainable=True
        )

    def call(self, inputs):
        return ops.matmul(inputs, self.w) + self.b

    def get_config(self):
        config = super().get_config()
        config.update({"units": self.units})
        return config


layer = Linear(64)
config = layer.get_config()
print(config)
new_layer = Linear.from_config(config)
```

If you need more flexibility when deserializing the layer from its config, you can also override the `from_config()` class method. This is the base implementation of `from_config()`:

```python
def from_config(cls, config):
    return cls(**config)
```

To learn more about serialization and saving, see the complete guide to saving and serializing models.

Privileged `training` argument in the `call()` method

Some layers, in particular the `BatchNormalization` layer and the `Dropout` layer, have different behaviors during training and inference. For such layers, it is standard practice to expose a `training` (boolean) argument in the `call()` method.

By exposing this argument in `call()`, you enable the built-in training and evaluation loops (e.g. `fit()`) to correctly use the layer in training and inference.

```

class CustomDropout(keras.layers.Layer):
    def __init__(self, rate, **kwargs):
        super().__init__(**kwargs)
        self.rate = rate
        self.seed_generator = keras.random.SeedGenerator(1337)

    def call(self, inputs, training=None):
        if training:
            return keras.random.dropout(
                inputs, rate=self.rate, seed=self.seed_generator
            )
        return inputs

```

Privileged `mask` argument in the `call()` method

The other privileged argument supported by `call()` is the `mask` argument.

You will find it in all Keras RNN layers. A mask is a boolean tensor (one boolean value per timestep in the input) used to skip certain input timesteps when processing timeseries data.

Keras will automatically pass the correct `mask` argument to `__call__()` for layers that support it, when a mask is generated by a prior layer. Mask-generating layers are the `Embedding` layer configured with `mask_zero=True`, and the `Masking` layer.

The `Model` class

In general, you will use the `Layer` class to define inner computation blocks, and will use the `Model` class to define the outer model -- the object you will train.

For instance, in a ResNet50 model, you would have several ResNet blocks subclassing `Layer`, and a single `Model` encompassing the entire ResNet50 network.

The `Model` class has the same API as `Layer`, with the following differences:

- It exposes built-in training, evaluation, and prediction loops (`model.fit()`, `model.evaluate()`, `model.predict()`).
- It exposes the list of its inner layers, via the `model.layers` property.
- It exposes saving and serialization APIs (`save()`, `save_weights()`...)

Effectively, the `Layer` class corresponds to what we refer to in the literature as a "layer" (as in "convolution layer" or "recurrent layer") or as a "block" (as in "ResNet block" or "Inception block").

Meanwhile, the `Model` class corresponds to what is referred to in the literature as a "model" (as in "deep learning model") or as a "network" (as in "deep neural network").

So if you're wondering, "should I use the `Layer` class or the `Model` class?", ask yourself: will I need to call `fit()` on it? Will I need to call `save()` on it? If so, go with `Model`. If not (either because your class is just a block in a bigger system, or because you are writing training & saving code yourself), use `Layer`.

For instance, we could take our mini-resnet example above, and use it to build a `Model` that we could train with `fit()`, and that we could save with `save_weights()`:

```
class ResNet(keras.Model):

    def __init__(self, num_classes=1000):
        super().__init__()
        self.block_1 = ResNetBlock()
        self.block_2 = ResNetBlock()
        self.global_pool = layers.GlobalAveragePooling2D()
        self.classifier = Dense(num_classes)

    def call(self, inputs):
        x = self.block_1(inputs)
```

```
        x = self.block_2(x)
        x = self.global_pool(x)
        return self.classifier(x)


resnet = ResNet()
dataset = ...
resnet.fit(dataset, epochs=10)
resnet.save(filepath.keras)
```

Putting it all together: an end-to-end example

Here's what you've learned so far:

- A `Layer` encapsulate a state (created in `__init__()` or `build()`) and some computation (defined in `call()`).
- Layers can be recursively nested to create new, bigger computation blocks.
- Layers are backend-agnostic as long as they only use Keras APIs. You can use backend-native APIs (such as `jax.numpy`, `torch.nn` or `tf.nn`), but then your layer will only be usable with that specific backend.
- Layers can create and track losses (typically regularization losses) via `add_loss()`.
- The outer container, the thing you want to train, is a `Model`. A `Model` is just like a `Layer`, but with added training and serialization utilities.

Let's put all of these things together into an end-to-end example: we're going to implement a Variational AutoEncoder (VAE) in a backend-agnostic fashion -- so that it runs the same with TensorFlow, JAX, and PyTorch. We'll train it on MNIST digits.

Our VAE will be a subclass of `Model`, built as a nested composition of layers that subclass `Layer`. It will feature a regularization loss (KL divergence).

```python

class Sampling(layers.Layer):
    """Uses (z_mean, z_log_var) to sample z, the vector encoding a digit."""

    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.seed_generator = keras.random.SeedGenerator(1337)

    def call(self, inputs):
        z_mean, z_log_var = inputs
        batch = ops.shape(z_mean)[0]
        dim = ops.shape(z_mean)[1]
        epsilon = keras.random.normal(shape=(batch, dim), seed=self.seed_generator)
        return z_mean + ops.exp(0.5 * z_log_var) * epsilon


class Encoder(layers.Layer):
    """Maps MNIST digits to a triplet (z_mean, z_log_var, z)."""

    def __init__(self, latent_dim=32, intermediate_dim=64, name="encoder", **kwargs):
        super().__init__(name=name, **kwargs)
        self.dense_proj = layers.Dense(intermediate_dim, activation="relu")
        self.dense_mean = layers.Dense(latent_dim)
        self.dense_log_var = layers.Dense(latent_dim)
        self.sampling = Sampling()

    def call(self, inputs):
        x = self.dense_proj(inputs)
        z_mean = self.dense_mean(x)
        z_log_var = self.dense_log_var(x)
        z = self.sampling((z_mean, z_log_var))
        return z_mean, z_log_var, z


class Decoder(layers.Layer):
    """Converts z, the encoded digit vector, back into a readable digit."""

    def __init__(self, original_dim, intermediate_dim=64, name="decoder", **kwargs):
        super().__init__(name=name, **kwargs)
        self.dense_proj = layers.Dense(intermediate_dim, activation="relu")
        self.dense_output = layers.Dense(original_dim, activation="sigmoid")

    def call(self, inputs):
```

```
        x = self.dense_proj(inputs)
        return self.dense_output(x)


class VariationalAutoEncoder(keras.Model):
    """Combines the encoder and decoder into an end-to-end model for training."""

    def __init__(
        self,
        original_dim,
        intermediate_dim=64,
        latent_dim=32,
        name="autoencoder",
        **kwargs
    ):
        super().__init__(name=name, **kwargs)
        self.original_dim = original_dim
        self.encoder = Encoder(latent_dim=latent_dim, intermediate_dim=intermediate_dim
        self.decoder = Decoder(original_dim, intermediate_dim=intermediate_dim)

    def call(self, inputs):
        z_mean, z_log_var, z = self.encoder(inputs)
        reconstructed = self.decoder(z)
        # Add KL divergence regularization loss.
        kl_loss = -0.5 * ops.mean(
            z_log_var - ops.square(z_mean) - ops.exp(z_log_var) + 1
        )
        self.add_loss(kl_loss)
        return reconstructed
```

Let's train it on MNIST using the `fit()` API:

```
(x_train, _), _ = keras.datasets.mnist.load_data()
x_train = x_train.reshape(60000, 784).astype("float32") / 255

original_dim = 784
vae = VariationalAutoEncoder(784, 64, 32)

optimizer = keras.optimizers.Adam(learning_rate=1e-3)
vae.compile(optimizer, loss=keras.losses.MeanSquaredError())

vae.fit(x_train, x_train, epochs=2, batch_size=64)
```

```
for i in range(100):
  print()
```